UNIVERSITY OF CALIFORNIA

SANTA CRUZ

ROUTING IN THE INTERNET USING PARTIAL LINK STATE INFORMATION

A dissertation submitted in partial satisfaction
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Marcelo Spohn

September 2001

The Dissertation of Marcelo Spohn
is approved:

Professor J.J. Garcia-Luna-Aceves, Chair

Professor Darrell Long

Professor Glen Langdon

Frank Talamantes
Vice Provost and Dean of Graduate Studies

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **SEP 2001**	2. REPORT TYPE	3. DATES COVERED **00-09-2001 to 00-09-2001**

4. TITLE AND SUBTITLE **Routing in the Internet Using Partial Link State Information**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Santa Cruz,Department of Computer Engineering,Santa Cruz,CA,95064**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**156**	

Contents

List of Figures v

List of Tables vii

Abstract viii

Acknowledgements x

Dedication xi

1 Introduction 1
- 1.1 Routing in Wired Networks 3
- 1.2 Routing in Wireless Networks 4
- 1.3 Organization of the Thesis 7

2 Adaptive Link-State Routing 9
- 2.1 Network Model 10
- 2.2 Operation of ALP 11
  - 2.2.1 Information Stored and Exchanged 14
  - 2.2.2 Validating Updates 15
  - 2.2.3 Processing Input Events 16
  - 2.2.4 Electing Designated Routers 22
- 2.3 ALP Correctness 24
- 2.4 Performance 31
- 2.5 Conclusions 34

3 Source Tree Routing 40
- 3.1 Updating Routes in Wireless Networks 42
- 3.2 STAR Description 45
  - 3.2.1 Network Model 45
  - 3.2.2 Overview 46
  - 3.2.3 Information Stored and Exchanged 48
  - 3.2.4 Validating Updates 49
  - 3.2.5 Exchanging Update Messages 51
  - 3.2.6 Impact of The Link Layer in LORA 57

3.2.7 Details on The Processing of Input Events 59
3.3 STAR Correctness 64
3.3.1 Correctness of STAR Under ORA 64
3.3.2 Correctness of STAR Under LORA 67
3.4 Performance Evaluation 70
3.4.1 Simulation Experiments 70
3.4.2 Comparison with Table-Driven Protocols 71
3.4.3 Comparison with DSR 74
3.4.4 Comparison with DSR Using Reliable Broadcasts 75
3.4.5 Comparison with DSR Using Rules LORA-1 to LORA-7 77
3.5 Conclusions 80

4 Neighborhood Aware Source Routing 83
4.1 NSR Description 85
4.1.1 Overview 85
4.1.2 Routing Information Maintained in NSR 87
4.1.3 Routing Information Exchanged by NSR 89
4.1.4 Operation of NSR 91
4.1.5 Using Neighbor IDs in Source Routes 102
4.2 Proof of Correctness 103
4.3 Performance Evaluation 108
4.3.1 Protocol Configuration 109
4.3.2 Comparison with STAR 110
4.3.3 Comparison with DSR 111
4.4 Conclusions 136

5 Summary and Future Work 137
5.1 Contributions 137
5.2 Future Work 140

Bibliography 142

List of Figures

2.1 Topology as seen by routers indicated with filled circle. Solid lines indicate links in source graph; dashed lines indicate links in topology graph but not in source graph. 13
2.2 ALP protocol building modules 14
2.3 Processing update message msg received from neighbor k 17
2.4 State diagram for a link l 19
2.5 Input events of the state diagram 20
2.6 Procedures used to determine to which neighbors a link-state update can be announced 24
2.7 Results for links changing cost 35
2.8 Results for links failing 36
2.9 Results for links recovering after failure 37
2.10 Results for nodes failing 38
2.11 Results for nodes recovering after failure 39

3.1 An example topology 47
3.2 Routers running STAR without the last part of LORA-3 being in effect. 54
3.3 Routers running STAR with the last part of LORA-3 being in effect. 55
3.4 The last part of LORA-3 not always triggers the generation of an update message: (a) network topology, and (b) source tree of node c after processing the failure of link (c, b). 56
3.5 STAR Specification 60
3.6 STAR Specification (cont.) 61
3.7 Cumulative distribution of packet delay experienced by data packets 81
3.8 Cumulative distribution of packet delay experienced by data packets 82

4.1 Failure of links (b, c) and (c, d) do not cause the generation of RERR packets when repairing the source route represented by the links in solid lines 87
4.2 Procedure to determine if X is greater than Y, where X and Y are derived from E_i or SN_i 94
4.3 Link-state information learned from processing RREQ and RREP packets . . . 96
4.4 Type of repairs that can be applied to a source route in a RREP packet 97
4.5 Type of repairs that can be applied to a source route in a DATA packet 99
4.6 Broken source-routes leading to transmission of RERR packets 100

4.7 Using *neighbor IDs* in source routes 102
4.8 Cumulative distribution function for the number of control packets generated . 116
4.9 The cumulative distribution function for the percentage of data packets received 117
4.10 The cumulative distribution function for the delay experienced by data packets 118
4.11 The cumulative distribution function for the number of hops traversed by data packets 119
4.12 Cumulative distribution function for the number of control packets generated . 120
4.13 A partial view of the cumulative distribution function for the number of control packets generated 121
4.14 The cumulative distribution function for the percentage of data packets received 122
4.15 The cumulative distribution function for the delay experienced by data packets 123
4.16 The cumulative distribution function for the number of hops traversed by data packets 124
4.17 Number of control packets transmitted using the Nsrc-Ndst pattern 125
4.18 Number of control packets transmitted using the Nsrc-1dst pattern 126
4.19 Number of control packets transmitted using the Nsrc-8dst pattern 127
4.20 Percentage of data packets received using the Nsrc-Ndst pattern 128
4.21 Percentage of data packets received using the Nsrc-1dst pattern 129
4.22 Percentage of data packets received using the Nsrc-8dst pattern 130
4.23 Number of control packets transmitted using the Nsrc-Ndst pattern 131
4.24 Number of control packets transmitted using the Nsrc-1dst pattern 132
4.25 Number of control packets transmitted using the Nsrc-8dst pattern 133
4.26 Percentage of data packets received using the Nsrc-Ndst pattern 134
4.27 Percentage of data packets received using the Nsrc-1dst pattern 135
4.28 Percentage of data packets received using the Nsrc-8dst pattern 136

List of Tables

3.1 Average performance of STAR, ALP, and TOB. 73

3.2 Average performance of STAR and DSR 76

3.3 Distribution of DATA packets delivered according to the number of hops traversed from the source to the destination 76

3.4 Performance of STAR and DSR 78

3.5 Number of hops traversed by data packets (pause time 0) 79

3.6 Changes in link connectivity 79

4.1 Number of control packets generated by NSR, STAR, and DSR in a 20-node network 111

4.2 Percentage of data packets delivered by NSR, STAR, and DSR in a 20-node network 111

Abstract

Routing in the Internet Using Partial Link State Information

by

Marcelo Spohn

This thesis focuses on routing in wired and wireless segments of the Internet using partial link-state information. Although efficient algorithms have been proposed based on both link-state and distance-vector information, link-state routing is more efficient than distance-vector routing when constraints are placed on the paths offered to destinations, which is the case for QoS routing offering paths with required delay, bandwidth, reliability, cost, or other parameters.

We present a new link-state routing protocol for wired internetworks called ALP (adaptive link-state protocol). In ALP, a router sends updates to its neighbors regarding the links in its preferred paths to destinations. Each router decides which links to report to its neighbors based on its local computation of preferred paths. A router running ALP does not ask its neighbors to delete links; instead, a router simply updates its neighbors with the most recent information about those links it decides to take out of its preferred paths.

We introduce and analyze two routing algorithms for wireless networks: the source-tree adaptive routing (STAR) protocol, and the neighborhood-aware source routing (NSR) protocol. STAR is the first example of a table-driven routing protocol that is more efficient than prior table-driven and on-demand routing protocols by exploiting link-state information to allow paths taken to destinations to deviate from the optimum in order to save bandwidth without creating loops. NSR is an on-demand routing protocol based on partial topology information and source routing. STAR is shown to be more efficient than the dynamic source routing (DSR) protocol in small ad hoc networks, and NSR is shown to outperform STAR and DSR in large wireless networks with mobile nodes.

Acknowledgements

I am indebted to many people for their support, teaching, ideas, encouragement, and criticism. The friendship and love of my wife, Luci, to whom this thesis is dedicated, was essential for the successful development of my graduate studies. Thanks “Lu”, for helping me learn.

My sincerest thanks to my advisor J.J. Garcia-Luna-Aceves for all the invaluable guidance and support he has given me since I arrived in Santa Cruz. I am immensely grateful to him for leading me to so many good paths, with no loops! His positive attitude, enthusiasm, good energy, and friendship are an inspiration.

I thank the members of my committee, Professors Glen Langdon and Darrell Long, for their advice and feedback. I would also like to thank Carol Mullane for her friendly help and advice.

A big thanks to my fellow “cocos” Brad Smith, Brian Levine, Chane Fullmer, Chris Parsa, Clay Shields, Ewerton Madruga, Hans-Peter Dommel, Jochen Behrens, Jyoti Raju, Lichun Bao, Lori Flynn, Mike Parsa, Rodrigo Garces, Soumya Roy, Srinivas Vutukury, and Tzamaloukas Assimakis for the friendship and intellectually estimulating research environment.

Thanks also to Dave Beyer, John Hight, Takayuki Kaiso, and Thane Frivold from Rooftop Communications, for their support with the simulation tool used in this thesis. It was a real pleasure to have the opportunity to work with such enthusiastic and sharp team.

I am particularly indebted to my parents, for all their love and hard work that allowed my dreams to come true. My grateful thanks mom and dad!

Special thanks are due to the CNPq, a Brazilian agency that funded part of this work. This work was also supported in part by the Defense Advanced Research Projects Agency (DARPA) under grant F30602-97-2-0338.

To my wife, Luci.

Chapter 1

Introduction

Routing is the network-layer function that selects the paths that data packets traverse from a source to a destination. The work on this thesis concentrates on routing based on the Internet Protocol (IP) [44].

The Internet is based on packet switching, which requires the use of *routing tables* specifying the next hops to destinations. These tables are maintained by means of distributed routing algorithms, which must adapt to resource failures and additions, and link cost changes caused by congestion, for example.

Routing algorithms can be categorized according to the way in which routers obtain routing information, and according to the type of information they use to compute preferred paths. In terms of the type of information used by routing protocols, routing protocols can be classified into link-state protocols and distance-vector protocols. Routers running a link-state protocol use topology information to make routing decisions; routers running a distance-vector protocol use distances and, in some cases, path information, to destinations to make routing decisions. In terms of the way in which routers obtain information, routing protocols have been classified as table-driven and on-demand.In an on-demand routing protocol, routers maintain path information for only those destinations that they need to contact as a source or relay of information. In a table-driven algorithm, each router maintains path information for each known destination in the network and updates its routing-table entries as needed.

Regardless of the way in which routers obtain routing information, or the type of information they use to compute preferred paths, or the type of network infrastructure they need to operate on, certain properties are desirable in a routing algorithm. Some of them are [52]:

- *Simplicity*: Simple algorithms are preferred for ease of implementation and higher efficiency in operational networks.
- *Robustness with respect to failures and changing conditions*: The algorithm must be able to adjust the routing decisions when traffic conditions change or when there is a resource failure. The algorithm monitors the network constantly and updates the routing information.
- *Stability of the routing decisions*: The routing algorithm should adapt smoothly to changes in operating conditions, i.e., a small change in operating conditions should provide a comparatively small change in routing decisions.
- *Fairness of the resource allocation*: Data flows with the same characteristics should result in similar packet delay and throughput.
- *Optimality of the packet travel times*: The routing algorithm should maximize the network designer's objective function, while satisfying design constraints.
- *Loop freedom*: At any instant, the paths implied from the routing tables of all hosts taken together should not have loops. Each router in the path from a source to a destination

should be visited only once.

- *Convergence characteristics*: The time required to converge after a topology change should not be high. This is required to maintain up-to-date network state information.
- *Processing and memory efficiency*: The resources used at each router should be minimal. The computation time spent at a node affects the convergence time of the routing algorithm.

In this dissertation, our objective is to satisfy most of the above mentioned attributes of routing algorithms, and address routing in wired and wireless networks.

1.1 Routing in Wired Networks

Prior work on routing protocols for wired internetworks is based on table-driven algorithms. Both distance-vector protocols (e.g., BGP [51], IDRP [50], RIP [26], and EIGRP [1]) and link-state protocols (e.g., ISO IS-IS [39] and OSPF [35]) are used in today's Internet. Several routing algorithms based on distance vectors have been proposed to eliminate the counting-to-infinity problem that prevents the Bellman-Ford algorithm from working efficiently in large networks (e.g., [18, 6, 47]) and a number of these algorithms have been shown to outperform the traditional approach used in implementing link-state routing. Most approaches to link-state routing are based on topology broadcast [17, 43]. Unfortunately, disseminating complete link-state information to all routers incurs excessive communication overhead. The link-vector algorithm (LVA) [19] was recently proposed to avoid the overhead of topology broadcast when using link-state information. In LVA, each router updates its neighbors with the state of each of the links it uses to reach a destination through one or more preferred paths, and also informs them of the links that it stops using to reach destinations. Updates to link states or deletions of links are propagated incrementally, based on the distributed computation of preferred paths at routers, just like distance information propagates in a distance-vector algorithm.

Although efficient algorithms have been proposed based on both link-state and distance-vector information, link-state routing is more efficient than distance-vector routing when constraints are placed on the paths offered to destinations, which is the case for QoS routing offering paths with required delay, bandwidth, reliability, cost, or other parameters. The communication overhead of a link-state protocol increases to the extent that more parameters have to be communicated for each link whose state is updated, i.e., the added overhead is at most linear with the number of link parameters. In contrast, the communication overhead of a distance-vector protocol grows with the number of combinations of values of the link parameters needed to define the quality of paths [28].

As the Internet continues to evolve to support QoS routing, obtaining more efficient approaches to link-state routing has become an important design and engineering problem. We present a new link-state routing protocol for wired internetworks called ALP (adaptive link-state protocol). In ALP, a router sends updates to its neighbors regarding the links in its preferred paths to destinations. Each router decides which links to report to its neighbors based on its local computation of preferred paths. In contrast to LVA, a router does not ask its neighbors to delete links; instead, a router simply updates its neighbors with the most recent information about those links it decides to take out of its preferred paths.

1.2 Routing in Wireless Networks

Multi-hop packet-radio networks, or ad hoc wireless networks, consist of mobile hosts interconnected by routers that can also move. The deployment of such routers is ad hoc and the topology of the network is very dynamic, because of host and router mobility, signal loss and interference, and power outages. In addition, the channel bandwidth available in ad hoc networks is relatively limited compared to wired networks, and untethered routers may need to operate with battery-life constraints.

Most routing algorithms for wireless ad hoc networks obtain routing information on an on-demand basis. The basic approach of an on-demand routing algorithm consists of allowing a router that does not know how to reach a destination to send a flood-search message to obtain the path information it needs. The first routing protocol of this type was proposed to establish virtual circuits in the MSE network [34], and there are several more recent examples of this approach (e.g., Ad Hoc On Demand Distance Vector (AODV) [42], Associativity-Based Routing (ABR) [55], Dynamic Source Routing (DSR) [30], Temporally-Ordered Routing Algorithm (TORA) [40], Signal Stability-Based Adaptive Routing (SSA) [13]). Source-tree bridges also use flood-search packets to obtain source routes from source to destination. Recently, the Dynamic Source Routing (DSR) protocol has been shown to outperform many other on-demand routing protocols [11]. On-demand routing protocols differ on the specific mechanisms used to disseminate flood-search packets and their responses, cache the information heard from other nodes' searches, determine the cost of a link, and determine the existence of a neighbor.

Examples of table-driven algorithms based on distance vectors are the routing protocol of the DARPA packet-radio network [31], the Destination-Sequenced Distance-Vector protocol (DSDV) [41], the Wireless Routing Protocol (WRP) [36], the Wireless Internet Routing Protocol (WIRP) [12], and least-resistance routing protocols [46]. Prior table-driven approaches to link-state routing in wireless networks are based on topology broadcast. However, disseminating complete link-state information to all routers incurs excessive communication overhead in an ad hoc network because of the dynamics of the network and the small bandwidth available. Accordingly, there are link-state routing approaches for packet-radio networks based on hierarchical routing schemes [49, 48, 10]. The Zone Routing Protocol (ZRP) [25] is a hybrid of on-demand and table-driven techniques.

A key issue in deciding which type of routing protocol is best for ad hoc networks is the communication overhead incurred by the protocol. Because data and control traffic share the same communication bandwidth in the network, and because untethered routers use the same energy source to transmit data and control packets, computing minimum-cost (e.g., least interference) paths to all destinations at the expense of considerable routing-update traffic is not practical in ad hoc networks with untethered nodes and dynamic topologies. The routing protocol used in an ad hoc network should incur as little communication overhead as possible to preserve battery life at untethered routers and to leave as much bandwidth as possible to data traffic.

To date, the debate on whether a table-driven or an on-demand routing approach is best for wireless networks has assumed that table-driven routing necessarily has to provide optimum (e.g., shortest-path) routing, when in fact on-demand routing protocols cannot ensure optimum paths. The Distance Routing Effect Algorithm for Mobility (DREAM) [14] was proposed to address the perceived limitations of prior on-demand and table-driven routing protocols. DREAM uses node coordinates rather than identifiers for routing. It disseminates coordinate information to all nodes and uses directed flooding to forward data packets to destinations. At each router, a data packet for a given destination is forwarded to all neighbor routers in the direction of the destination. Another routing approach based on location information is the Location-Aided Routing (LAR) protocol [33]. LAR is an on-demand routing protocol that uses location information to reduce the scope of the flood search needed to obtain a route to a destination.

We introduce and analyze two routing algorithms for wireless networks: the source-tree adaptive routing (STAR) protocol, and the neighborhood-aware source routing (NSR) protocol. STAR is the first example of a table-driven routing protocol that is more efficient than prior table-driven and on-demand routing protocols by exploiting link-state information to allow paths taken to destinations to deviate from the optimum in order to save bandwidth without creating loops. NSR is an on-demand routing protocol based on partial topology information and source routing.

1.3 Organization of the Thesis

This thesis is organized as follows:

- **Chapter 2** presents ALP, a link-state table-driven routing protocol based on partial topology information. We show through simulations that ALP has better performance than the state of the art routing algorithms used in today's wired Internet.

- **Chapter 3** describes STAR, a link-state table-driven routing protocol based on partial topology information suitable for wireless mobile networks. Two variants of STAR are investigated. In the first variant, the routing protocol attempts to update routing tables as quickly as possible to provide paths that are optimum with respect to a defined metric. In contrast, in the second variant, the routing protocol attempts to provide viable paths according to a given performance metric, which need not be optimum, to incur the least amount of control traffic.

- **Chapter 4** presents NSR, an on-demand routing protocol based on partial link-state information that scales well in wireless mobile networks, outperforming Link-Cache DSR, the best performing routing protocol.

- **Chapter 5** gives a summary of this work, together with some conclusions and directions for future research.

Chapter 2

Adaptive Link-State Routing

In the adaptive link-state protocol (ALP), a router sends updates to its neighbors regarding the links in its preferred paths to destinations. Each router decides which links to report to its neighbors based on its local computation of preferred paths.

Because routers have different topology maps, routers may have to erase the records of links that are no longer used, so that no router in the network attempts to use any link on the basis of old link-state information distributed about that link when it was being used by other routers. In all prior link-state protocols, any given link has only two local states: a router either has or does not have a record for the link. In contrast, ALP enables the partial dissemination of link-state information by assigning one of three different labels to any link record. A router may not have a local record of a link or decides to erase its local record of the link (an implicit label of 0), or may be using the link to reach some destination (label 1), or may have stoped using the link to reach a destination but has not asked its neighbors to forget about the link (label 2).

ALP validates link states using time stamps and a router accepts only more recent link-state updates. The state of failed links is erased by aging only. Furthermore, when multiple routers are connected through a broadcast medium (e.g., a LAN), they elect distributedly a designated router for each link reported over the broadcast medium; this reduces the number of updates per link sent over a given network.

Unlike OSPF or any of the hierarchical link-state routing schemes proposed to date [10], ALP does not require backbones, the dissemination of complete cluster topology within a cluster, or the dissemination of the complete inter-cluster connectivity among clusters. Furthermore, ALP can be used with distributed hierarchical routing schemes proposed in the past for both distance-vector or link-state routing [32, 10, 38, 2]. Because routers in ALP propagate link-state information selectively, it incurs less communication overhead than algorithms based on topology broadcast.

The following sections introduce the network model assumed throughout the rest of the chapter, describe ALP, show that ALP converges to correct paths a finite time after the occurrence of an arbitrary sequence of link-cost or topological changes, calculate its complexity, and present simulation results comparing ALP's performance against the performance of an ideal topology-broadcast algorithm, the distributed Bellman-Ford algorithm (DBF), and LVA.

2.1 Network Model

In ALP, routers maintain a partial topology map of their network. In this study we focus on flat topologies only, i.e., there is no aggregation of topology information into areas or clusters.

The topology of a network is modeled as a directed graph $G = (V, E)$, where V is the set of nodes and E is the set of edges connecting the nodes. Each node has a unique identifier and represents a router with input and output queues of unlimited capacity updated according to a FIFO policy. For the purpose of routing-table updating, a Node A can consider another Node B to be adjacent (we call such a node a “neighbor”) if there is link-level connectivity between A and B and A receives update messages from B reliably. Accordingly, we map a physical broadcast link connecting multiple nodes into multiple point-to-point bidirectional links defined for these nodes. A functional bidirectional link between two nodes is represented by a pair of edges, one in each direction and with a cost associated that can vary in time but is always positive.

An underlying protocol, which we call the neighbor protocol, assures that a router detects within a finite time the existence of a new neighbor, the loss of connectivity with a neighbor, and the reliable transmission of packets between neighbors. All messages, changes in the cost of a link, link failures, and new-neighbor notifications are processed one at a time within a finite time and in the order in which they are detected. Because of the neighbor protocol, ALP assumes that all messages transmitted over an operational link are received correctly and in the proper sequence within a finite time. Routers are assumed to operate correctly, and information is assumed to be stored without errors.

2.2 Operation of ALP

In ALP, each router reports to its neighbors the characteristics of every link it uses to reach a destination through a preferred path. The set of links used by a router in its preferred paths is called the *source graph* of the router. A router knows its adjacent links and the source graphs reported by its neighbors; the aggregation of a router’s adjacent links and the source graphs reported by its neighbors constitute a partial *topology graph.* The links in the source graph and topology graph must be adjacent links or links reported by at least one neighbor. The router uses one or more local *route selection algorithms*, the topology graph to generate its own source graph, and a routing table specifying the successor, successors, or paths to each destination.

The basic update unit used in ALP to communicate changes to source graphs is the link-state update (LSU). An LSU reports the characteristics of a link; an update message contains one or more LSUs. For a link between router u and router or destination v, router u is called the *head node* of the link in the u to v direction. The head node of a link is the only router that can report changes in the parameters of that link.

The head of a link reports the state of the link in an LSU if it uses the link to reach any destination, i.e., if the link is in its source graph. The LSUs from a router specify the state of links that the router currently uses in its source graph and links that are removed from its topology graph that had been recently used in its source graph. Hence, a router in ALP always tells its neighbors about the links it uses to reach destinations, and does not tell its neighbors about links it stops using to reach destinations, unless it is necessary to prevent old reports of such links to be taken as valid.

More specifically, a router sends LSUs about a link in the following cases: (a) when an LSU is received for the link, or the link changes state, causing the link to be added to the router's source graph; (b) when the link is already in the router's source graph and a more recent LSU is received for the link, or the link changes state; and (c) when the link is not in the router's source graph but has been in the source graph before, and an LSU is received for any link or a topology change occurs that forces the link to be deleted from the router's topology graph.

As we show in subsequent sections, this method of sending LSUs ensures that, within a finite time, all routers in the network have consistent routing information, and in terms of update messages generated is close to the minimum communication overhead possible.

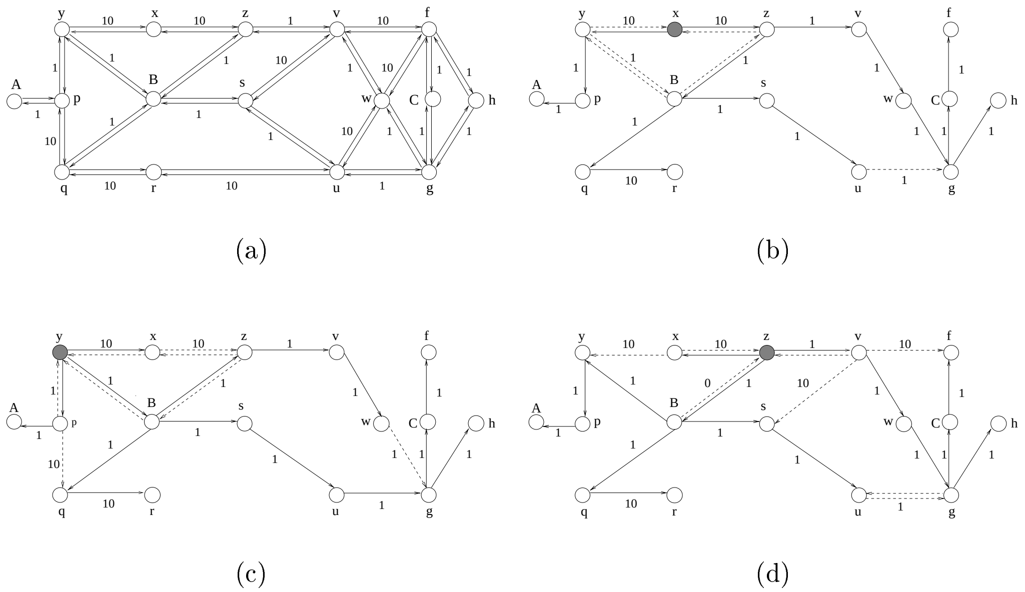


Figure 2.1: Topology as seen by routers indicated with filled circle. Solid lines indicate links in source graph; dashed lines indicate links in topology graph but not in source graph.

Figure 2.1 illustrates the fact that routers in ALP have to maintain only partial topology information. For simplicity, this figure and the rest of the paper assume that a single parameter is used to characterize a link in one of its directions, which we will call the cost of the directed link. Furthermore, although any number and type of local route selection algorithms can be used in ALP, we describe ALP assuming that shortest paths are used for routing and that Dijkstra's shortest-path first is used locally at each router. Figure 2.1b through 2.1d show the selected topology according to ALP at the routers indicated with filled circles. Solid lines represent the links that are part of the source graph of the respective router, dashed lines represent links that are part of the router's topology graph but not of its source graph. Arrowheads on links indicate the direction of the link stored in the router's topology graph. Router x's source graph shown in Figure 2.1b is formed by the source graphs reported by its neighbors y and z, and the links for which router x is the head node (namely links (x,y) and (x,z)). From the figure, the savings in storage requirements are clear, even for the small

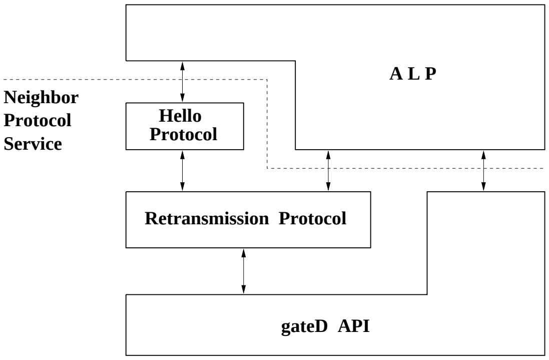


Figure 2.2: ALP protocol building modules

example shown in the figure.

We have developed a routing protocol framework based on the API provided by gateD [8] for implementation of routing protocols (Figure 2.2). A Hello Protocol is used to detect the presence of new neighbors and the loss of connectivity to them; the Retransmission Protocol is responsible for delivering update messages correctly and in the proper sequence to neighbors, as long as the physical link and network interface are operational. Among other things, the gateD API manages the routing forwarding table and relays indications to ALP reporting changes in the parameters of adjacent links, such as link cost changes, link failures, and link recoveries. In the rest of this paper we describe ALP assuming the services provided by the three underlying modules, which corresponds to the services provided by the neighbor protocol.

2.2.1 Information Stored and Exchanged

The LSU for a link (u, v) in an update message is a tuple (u, v, l, ts, del) reporting the characteristics of the link, where l represents the cost of the link, ts is the timestamp assigned to the LSU, and del is a flag set to 1 if the LSU is a DELETE.

In our description, we refer to an LSU that has a cost infinity and the del field equal to zero as a RESET, and refer to an LSU with an infinity link cost and the *del* field set to one, as a DELETE.

A router i maintains a topology graph TG_i, a source graph SG_i, a routing table, and the set of neighbors N_i. The record entry for link (u, v) in the topology graph of router i is denoted $TG_i(u, v)$ and is defined by the tuple $(u, v, l, ts, age, F, l', tag, rn)$, and a parameter p in the tuple is denoted by $TG_i(u, v).p$.

$TG_i(u, v).F$ contains the set of network interfaces through which node i has received up-to-date link-state information for (u, v), and $TG_i(u, v).F(f)$ holds the addresses of the neighbors who have reported up-to-date link-state information for (u, v) through interface f. In the example shown in Figure 2.1, router x's topology graph would have a record for link (s, u) indicating that y and z reported the same link. The link parameters l', tag, and rn are described in the next sections.

A vertex v in TG_i is denoted $TG_i(v)$ and contains a tuple $(d, pred)$ whose values are used on the computation of the source graph. $TG_i(v).d$ is the distance of the path $i \rightsquigarrow v$, and $TG_i(v).pred$ is v's predecessor in $i \rightsquigarrow v$.

The source graph SG_i is a subset of TG_i. The routing table contains record entries for destinations in SG_i, each entry consists of the destination address, the cost of the path to the destination, and the address of the next-hop towards the destination.

2.2.2 Validating Updates

Because of delays in the routers and links of an internetwork, update messages sent by a router may propagate at different speeds along different paths. Therefore, a given router may receive an LSU from a neighbor with stale link-state information, and a distributed termination-detection mechanism is necessary for a router to ascertain when a given LSU is valid and avoid the possibility of LSUs circulating forever. ALP implements the termination-detection mechanism used in several prior link-state protocols based on topology broadcast [43], which consists in time stamps.

A router receiving an LSU accepts the LSU as valid if the received LSU has a larger timestamp than the timestamp of the LSU stored from the same source, or if there is no entry for the link in the topology graph. There is a special case in which a router other than the head of the link can change the cost of a link to infinity and report the new cost to the neighbors; this type of LSU will be considered valid under certain circumstances, as discussed in the next section. Each LSU sent by the same source specifies the current timestamp. Alternatively, a large linear sequence number space can be used, together with a reset mechanism for the sequence number to guard against malfunctions [35]. We opt for the timestamp method in order to make our treatment of ALP simple.

2.2.3 Processing Input Events

An update message from a router k consists of a list of LSUs reporting incremental updates to its source graph and deletion of links from the topology graph not caused by aging; the procedure *Update* (Figure 2.3) is executed when a router i processes an update message. First, the topology graph is updated, then the source graph is updated, which may cause the router to update its routing table and to send its own update message.

An LSU for (u, v) updates the topology graph if its timestamp is larger than the timestamp maintained for the same link in the topology graph, or no entry for the link exists in the topology graph, or the entry in the topology graph has the cost set to infinity and the LSU has the same timestamp as the entry in the topology graph but the link cost is not infinity.

An LSU is considered outdated not only if it specifies a timestamp that is smaller than

```
Update($k, msg$)
{
    Update_Topology_Graph($k, msg$);
    $newSG$ ← Build_Shortest_Path_Tree();
    Process_Cost_Increase_State_1($newSG$);
    Process_Cost_Increase_State_2($newSG$);
    Process_Links_RemovedFrom_SG($newSG$);
    event ← Update_Routing_Table($newSG$);

    if ( event = NEW_LINK or
         event = PARAMETER_CHANGE or
         event = NEWSG_EMPTY )
    {
        Compare_Source_Graphs($SG_i, newSG$);
    }

    $SG_i$ ← $newSG$;

    if ( $k \neq i$ )
        Send();
}
```

Figure 2.3: Processing update message *msg* received from neighbor k

the one in the topology graph, but also if the timestamps are the same and the LSU carries a link cost infinity, while the entry in the topology graph has a cost different than infinity and the link is in the source graph. The reception of an outdated LSU causes the router to send an LSU with up-to-date information to the neighbor that originated the update message.

If the LSU is a valid RESET and there is an entry in the topology graph for the link, the LSU is forwarded to the neighbors.

A new source graph is computed and the routing table is updated if new link-state information is added to the topology graph or links are deleted from the topology graph. The shortest-path tree is generated by running Dijkstra SPF algorithm on the topology graph.

Rather than generating delete updates every time a link is removed from the source graph, as is the case in LVA, ALP reports to its neighbors the new value of the parameters of a link removed from the source graph if the cost of the link has increased. For those links that are removed from the source graph and that had not a cost increase, the node will only announce their removal when it learns that the cost of such links increased. The links stored in the topology graph have a tag that is used to keep track of those links that had the source graph removal announcement postponed, and gives the current state of the link in the state diagram of Figure 2.4. The state diagram shows the transition to a new state for a link $l = (u, v)$, given its current state and the type of input event received for the link. The tag of a link (u, v) for node i is denoted $TT_i(u, v).tag$, and its possible values at time t are the following:

0: Link (u, v) is not in the source graph at time t_i, where $t_{reset} \leq t_i \leq t$ and t_{reset} is the time the link last transitioned to state 0. The tag of a link is set to 0 when the link is inserted into the topology graph or when the cost of the link increases.

1: Link (u, v) is in the source graph at time t.

2: Link (u, v) is not in the source graph at time t, but it was in the source graph at time t_i, where $t_{reset} \leq t_i < t$. The removal of the link from the source graph had not been announced.

The transition to NIL in the state diagram corresponds to the deletion of the link from the topology graph. The description of possible input events summarized on the state diagram in Figure 2.4 are given in Figure 2.5.

In our current implementation of ALP, a link in the topology graph has just one *reporting neighbor*. This contrasts with LVA, which considers a reporting neighbor to be any neighbor that has reported an LSU with sequence number that matches the sequence number

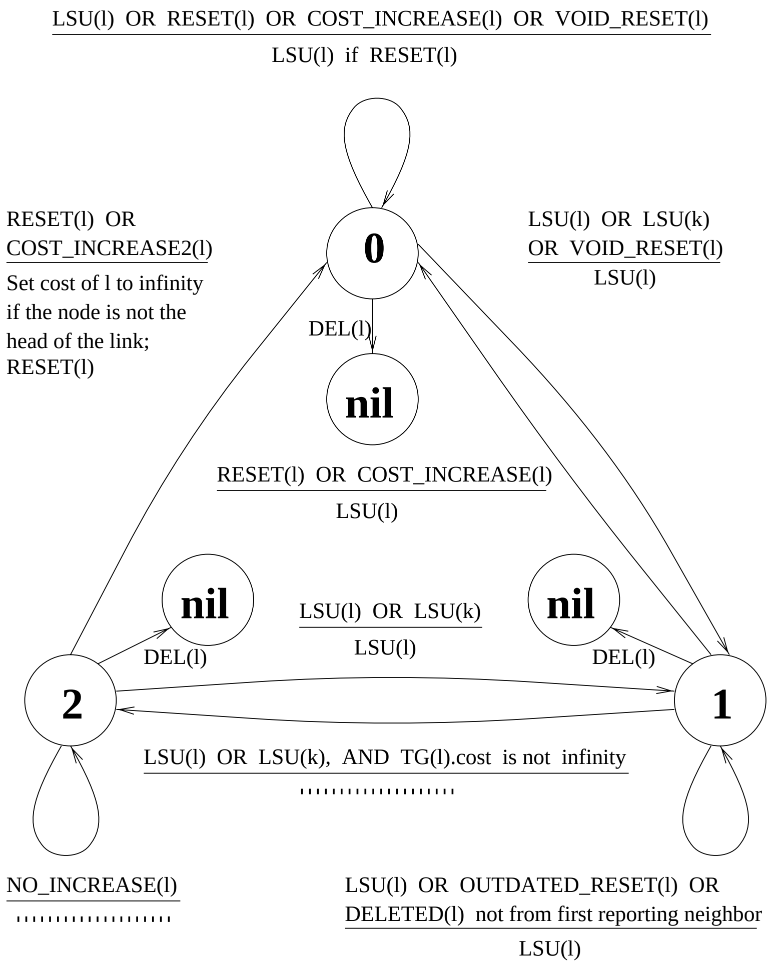

LSU(l) OR RESET(l) OR COST_INCREASE(l) OR VOID_RESET(l)
LSU(l) if RESET(l)
RESET(l) OR
COST_INCREASE2(l)
Set cost of l to infinity
if the node is not the
head of the link;
RESET(l)
0
LSU(l) OR LSU(k)
OR VOID_RESET(l)
LSU(l)
DEL(l)
nil
RESET(l) OR COST_INCREASE(l)
LSU(l)
nil
LSU(l) OR LSU(k)
LSU(l)
nil
DEL(l)
DEL(l)
2
1
LSU(l) OR LSU(k), AND TG(l).cost is not infinity
NO_INCREASE(l)
LSU(l) OR OUTDATED_RESET(l) OR
DELETED(l) not from first reporting neighbor
LSU(l)

Figure 2.4: State diagram for a link l

for the link in the topology graph. The reporting neighbor for a link (u, v) in the topology graph is denoted $TT_i(u, v).rn$, and consists of the address of the neighbor that last reported a valid LSU for the link if the state of (u, v) is 0, or the address of the neighbor that is in the shortest-path to u if (u, v) is in the source graph (i.e, the state of (u, v) is 1), or the address of the neighbor that was in the shortest-path to u at the time link (u, v) was removed from the source graph and transitioned to State 2. The reporting neighbor of a neighbor's adjacent link is the neighbor itself.

LSU(l) :	LSU for link l other than RESET(l), VOID_RESET(l), OUTDATED_RESET(l), COST_INCREASE(l), COST_INCREASE2(l), and DELETE(l).
RESET(l) :	LSU with cost infinity generated when link l fails or when l transitions from State 2 to State 0.
VOID_RESET(l) :	Timestamp of LSU is equal to the timestamp of the link in the topology graph, the cost of the LSU is not infinity, and the cost of the link in the topology graph is infinity.
OUTDATED_RESET(l) :	Timestamp of link in LSU is equal to the timestamp of the link in the topology graph, the cost of the LSU is infinity, the *del* field of the LSU is zero, and link l is in the source graph.
COST_INCREASE(l) :	Cost of link l has increased.
COST_INCREASE2(l) :	Cost of LSU is greater than $TT_i(l).l'$.
DELETE(l) :	LSU reporting link l was removed from the topology graph.
DEL(l) :	DELETE(l) or there is no reporting neighbor for link l.
NO_INCREASE(l) :	cost of link l in LSU is not greater than $TT_i(l).l'$

Figure 2.5: Input events of the state diagram

When a link (u, v) is removed from the source graph SG_i and transitions to State 2, node i needs to store the current cost of the link in $TT_i(u, v).l'$, which is used for checking increases in the cost of the link while (u, v) is in State 2.

After computing the new source graph $newSG$ (Figure 2.3) the router generates link-state updates for those links whose cost has increased and were removed from the source graph, i.e., the links have transitioned from State 1 to State 0 (Figure 2.4). Then, the router generates RESETs for those links that had the cost increased while in State 2. If the router that transitions a link from State 2 to State 0 is not the head of the link, the cost of the link in the topology graph is set to infinity. Procedure *Update* then executes *Process_Links_RemovedFrom_SG* which sets $TT_i(u, v).l' \leftarrow TT_i(u, v).l$ and $TT_i(u, v).tag \leftarrow 2$ for each link (u, v) that was removed from the source graph and had not the cost increased. The router then compares the new source graph $newSG$ against the current source graph SG_i, and LSUs are created with the link-state information for links that are in $newSG$ but not in SG_i, or that are in both graphs but had their timestamp changed. After LSUs are generated, $newSG$ becomes the current source graph SG_i.

If a link cost changes, then its head node is notified by an underlying protocol. The router then runs *Update* with the appropriate message as input; the LSU in the message gets a current timestamp. This holds for simple changes in link cost, as well as for a link failure. The same approach is used for a new link or a link that comes up after a failure. When a router establishes connectivity to a new neighbor, the router sends its complete source graph to the neighbor (much like a distance vector protocol sends its complete routing table).

When the link (i, k) to neighbor k fails, the topology graph is updated to erase the neighbor from the set $TT_i(i, k).F(f)$, and a RESET for (i, k) is transmitted to the neighbors. For each link (u, v) in the topology graph whose reporting neighbor is router k, router i generates a DELETE update for (u, v) and deletes the link from the topology graph. A router that receives a DELETE update from a node other than the reporting neighbor transmits to the sender of the DELETE an LSU with the current state of the link if the link is in the source graph. This guarantees that the tree of reporting neighbors for link $(u, v) \in E$, formed by the links $(i, TT_i(u, v).rn) \in E$, for each node $i \in V$, is updated accordingly. Link-state information for failed links that have a reporting neighbor must be kept in the topology graph in order to validate incoming LSUs for the link.

Consider the topology in Figure 2.1 and assume that link (y, B) increases its cost dramatically (e.g., from 1 to 100). Node y processes the link-cost increase and generates a new source graph; the update message sent by node y to its neighbors specifies an LSU for the link (y, B) with the new cost and LSUs for links (p, q), (x, z), (z, B) and (w, g), which must now be used to reach all the nodes in the graph. Note that router y does not inform its neighbors that it removed links (B, z), (B, q) and (u, g) from its source graph, as would be the case in LVA [19], but makes the links to transition from State 1 to State 2 (see Figure 2.4). An important difference between ALP and LVA is that in ALP a router informs its neighbors of a link removed from its source graph only if it is removed because its cost increased or because there is no reporting neighbor for the link anymore (in which case the cost is set to infinity). LVA reports all deletions to the source graph, and all such deletions represent infinite link costs.

2.2.4 Electing Designated Routers

Because routers in ALP communicate partial topology information to their neighbors, defining a designated router as in OSPF to be in charge of sending topology information over a network connecting multiple neighbor routers cannot be applied. In ALP, a link is assigned a designated router if it needs to be reported by at least one router over a given broadcast medium (a LAN, a network, or a link).

The idea of assigning one designated router per link-state update consists of making only one router responsible for reporting the link-state update in a broadcast link. In this way, when an adjacency is formed with a new neighbor x through a broadcast link, x will receive only one copy of the link-state information for a link (u, v) from the routers that already have adjacencies in the link. To accomplish this, the topology graph entry $TG_i(u, v)$ maintains the set of interfaces through which node i has received link-state updates for the link (u, v), as well as the list of neighbors attached to the interface that have reported the link-state update. $TG_i(u, v).F$ contains the set of interfaces, and $TG_i(u, v).F(f)$ is the list of neighbors with adjacencies through interface f that have reported a link-state update.

When node i receives a valid LSU from neighbor x for link (u, v), the LSU for

(u, v) is forwarded to all neighbors except x, i then stores neighbor's x's address in the list $TG_i(u,v).F(f)$

```
$TG_i(u,v).F \leftarrow \emptyset$;
$TG_i(u,v).F \leftarrow TG_i(u,v).F \cup \{f\}$;
$TG_i(u,v).F(f) \leftarrow \{$ address of $x\}$;
```

If the received LSU (u, v) is not valid, but its timestamp is equal to the one stored in the topology graph, node i also stores neighbor's x's address in the list $TG_i(u,v).F(f)$, where f is the incoming interface

```
if ( $f \notin TG_i(u,v).F$ )
  $TG_i(u,v).F \leftarrow TG_i(u,v).F \cup \{f\}$;
$TG_i(u,v).F(f) \leftarrow$ SORT($TG_i(u,v).F(f) \cup \{$ address of $x\}$);
```

When a router i reports an LSU (u, v) through interface f, it adds the address of f into $TG_i(u,v).F(f)$.

The list of neighbors $TG_i(u,v).F(f)$ is always sorted in ascending order of router addresses. The first router address in the list $(TG_i(u,v).F(f).0)$ is the one with the smallest address.

When a new adjacency is formed to a neighbor k through interface f, node i will only report an LSU for link (u, v) to k if i is the head node of the link, or $f \in TG_i(u,v).F$ and $TG_i(u,v).F(f).0$ equals f's address. The procedure *DST_SET* shown in Figure 2.6 returns the set of interfaces through which a link-state update for link (u, v) can be announced, and *ANNOUNCE* returns TRUE if node i can announce an LSU for (u, v) through the interface i has connectivity to neighbor k. For any given link used by a set of routers connected to a LAN, the router with the smallest ID is the only one allowed to send LSUs for the link over the LAN.

```
DST_SET$(u, v)$
{
    $F \leftarrow$ set of operational interfaces;

    if ( $r \neq i$ )
        for each ( interface $f \in TG_i(u,v).F$ )
            if ( $TG_i(u,v).F(f) \neq \emptyset$ and $TG_i(u,v).F(f).0 \neq f.address$ )
                $F \leftarrow F - \{f\}$;

    return $F$;
}

ANNOUNCE$(k, u, v)$
{
    $announce \leftarrow$ TRUE;
    $f \leftarrow$ interface attached to $k$;

    if ( $r \neq i$ and $f \in TG_i(u,v).F$ )
        if ( $TG_i(u,v).F(f) \neq \emptyset$ and $TG_i(u,v).F(f).0 \neq f.address$ )
            $announce \leftarrow$ FALSE;

    return $announce$;
}
```

Figure 2.6: Procedures used to determine to which neighbors a link-state update can be announced

When i detects loss of connectivity to a neighbor x attached to a broadcast link through interface f, and x was the only router in $TG_i(u,v).F(f)$, router i will announce an LSU for (u,v) in the broadcast link if it has a path to destination v whose successor is not a neighbor in the broadcast link. This guarantees that new neighbors that have not received link-state information about (u,v) will get it as soon as i detects lack of connectivity to x.

2.3 ALP Correctness

In this section we show that routers executing ALP stop disseminating link-state updates and obtain shortest paths to destinations within a finite time after the cost of one or more links changes and there are no more changes afterwards.

For simplicity of exposition, we assume that all links are bidirectional point-to-point links and that shortest-path routing is implemented. Let t_0 be the time when the last of a finite number of link-cost changes occur, after which no more such changes occurs. The network $G = (V, E)$ in which ALP is executed has a finite number of nodes ($\mid V \mid$) and links ($\mid E \mid$), and every message exchanged between any two routers is received correctly within a finite time. According to ALP's operation, for each direction of a link in G, there is a router that detects any change in the cost of the link within a finite time.

The following theorems relay on the use of timestamps as described in Section 2.2.2; the same approach applies if an alternative update validation scheme based on resets is used. We also assume that all routers use the same type of tie-braking rules in computing shortest paths, e.g., if a shortest path to j is obtained through two different relays, routers choose the relay with the smallest identifier.

Lemma 1 *The dissemination of LSUs in ALP, other than DELETEs, stops a finite time after* t_0.

Proof: A router that detects a change in the cost of any outgoing link must update its topology graph, update its source graph as needed, and send an LSU if the link is added to or is updated in its source graph. Let l be the link that last experiences a cost change up to t_0, and let t_l be the time when the head of link l originates the last LSU of the sequence of LSUs originated as a result of the link-cost change occurring up to t_0. Any router that receives the LSU for link l originated at t_l must process the LSU within a finite time, and decides whether or not to forward the LSU based on its updates to its source graph. A router can accept and propagate an LSU only once because each LSU has a timestamp; accordingly, given that G is finite, there can only be a finite chain of routers that can propagate the LSU for link l originated at t_l, and the same applies to any LSU originated from the finite number of link-cost changes that occur up to t_0. Therefore, ALP stops the dissemination of LSUs a finite time after t_0. □

Lemma 2 *The dissemination of DELETEs in ALP stops a finite time after t_0.*

Proof: A router i that detects failure of the link to the reporting neighbor of a link l in the topology graph must delete l from the topology graph, update its source graph, and send a DELETE LSU for link l. Let the failed link be the link that last experiences a cost change up to t_0, and let t_l be the time when node i originates the last LSU of the sequence of LSUs originated as a result of the link-cost changes occurring up to t_0. Any router that receives the DELETE for link l originated at t_l must process the DELETE within a finite time, and forwards the DELETE after deleting l from its topology graph if the sender of the DELETE was the first reporting neighbor of l. A router can accept and propagate a DELETE only once because the link is deleted from the topology graph when the DELETE is accepted for the first time, and a DELETE for link l is not propagated if link l is not in the topology graph of the router processing the DELETE. Given that G is finite, there can only be a finite chain of routers that propagate the DELETE for link l originated at t_l, and the same applies to any DELETE originated from the finite number of link-cost changes that occur up to t_0. Therefore, ALP stops the dissemination of DELETEs a finite time after t_0. □

Theorem 1 *The dissemination of LSUs in ALP stops a finite time after t_0.*

Proof: The proof is immediate from Lemmas 1 and 2. □

From Theorem 1, it must be true that there is a time t_s when no more LSUs are queued or in transit anywhere in the network.

Lemma 3 *A router with a tag value of 1 for link l at time t_s must be the head of the link or have at least one neighbor with a tag value of 2 or 1.*

Proof: The proof is obvious if the router is the head of the link. Assume that router i is not the head of link l and that all of its neighbors have tags equal to 0 at time t_s.

Because router i is not the head of link l and has link l in its source graph, it must have received an LSU reporting l from at least one neighbor k at some time $t' < t_s$, which required k to have link l in its source graph at that time, i.e., to have a tag value of 1 for l at time $t' < t_s$. By assumption, k has a tag equal to 0 for link l, which means that k must have transitioned its tag value from 1 or 2 to 0 before time t_s. According to ALP's operation, at the time of its transition, k must have sent an LSU reporting an increase in the cost of link l, and it may also have sent LSUs for links that k adds or updates in its source graph. Because by assumption no LSUs are queued at or in transit to router i at time t_s, i must have processed the LSU from k indicating the cost increase for l, as well as any LSUs needed to bring i topology graph consistent with k's source graph.

Because none of i's neighbors use link l in their shortest paths, because i has received the LSUs from k that exclude link l from being part of any shortest path from k, and because all routers use the same tie-braking rules for shortest paths, it follows that router i cannot use l in any of its shortest paths, because k does not. Accordingly, router i must transition to a tag value of 0 or 2 after processing the LSUs from k, and the Lemma is true. □

Lemma 4 *A router with a tag value of 2 for link l at time t_s must be the head of the link or have at least one neighbor with a tag value of 2 or 1.*

Proof: The proof is obvious if i is the head of link l, because i may have shorter paths to the tail of the link than the link itself. Assume that router i is not the head of link l and that all its neighbors have tags equal to 0 for link l at time t_s.

Because router i is not the head of link l and has link l in its source graph, it must have received an LSU reporting l from at least one neighbor k at some time $t' < t_s$. Following the same line of argument used in the proof of Lemma 3, we can show that, at time t_s, router i must have processed the LSU from k indicating the cost increase for l, together with any LSUs needed to bring i topology graph consistent with k's source graph. According to ALP's operation, when router i has a tag value of 2 for link l and receives an LSU reporting a cost increase for l, then it must transition to a tag value of 0 and send an LSU; therefore, the Theorem is true. □

Theorem 2 *In a connected network, and in the absence of link failures, all routers have the most up-to-date link-states they need to compute shortest paths to all destinations within a finite time after t_s.*

Proof: The proof is by induction on the number of hops of a shortest path to a destination, and is basically a generalization of the proof for SPTA [3].

Consider the shortest path from router s_0 to a destination j at time t_s, and let h be the number of hops along such a path. For $h = 1$, the path from s_0 to j consists of one of the router's outgoing links. By assumption, an underlying neighbor protocol provides the correct parameter values of adjacent links within a finite time; therefore, the Theorem is true for $h = 1$, i.e., s_0 must have link (s_0, j) in its source graph, which means that its tag value for the link is 1 and it must have sent its neighbors an LSU for that link.

Assume that that any router with a path of n or fewer hops to j has the correct link-state information about all the links in the shortest path to j, and consider the case in which the path from s_0 to j at time t_s is $n + 1$ hops.

Router s_0 has a tag value of 1 for each link in the shortest path to j, because the path belongs to its source graph. For any such link l in the shortest path to j, it follows from Lemma 3 that the router has a neighbor that by time t_s has reported an LSU it can believe that specifies the up-to-date cost of l. Accordingly, the shortest path from s_0 to j must be through a neighbor s_1 with a tag value of 1 or 2 for link l, which means that s_1 must send the most up-to-date LSUs it receives for each link in its shortest path to j. The sub-path from s_1 to j has $h-1$ hops and, by the inductive assumption we have made, such a path must be the true shortest path from s_1 to j by time t_s. Because all routers use the same tie-braking rules to choose shortest paths, this also means that that s_1 must have a tag value of 1 for each link in its shortest path to j.

Because it is also true that s_0 has the most recent link-state information about link (s_0, s_1), it follows that s_0 has the most recent information about all the links in its chosen path to j. The Theorem is therefore true, because the same argument applies to any chosen destination and router. □

Theorem 3 *In a connected network, and in the absence of link failures, a tree of reporting neighbors for a link l will be formed within a finite time after t_s.*

Proof: For the neighbors of the head of the link l the root of the tree of reporting neighbors is the head of the link. The tree of reporting neighbors consist of routers whose value of the tag for link l can be 0, 1, or 2. Routers that have the tag set to 1 elect as the reporting neighbor for l the next hop in the shortest-path to the head of the link. Given that the source graph is computed within a finite time after t_s according to Theorem 2, the subtree of the tree of reporting neighbors that includes the source graph is computed a finite time after t_s. Whenever a valid link-state update for l is processed and l has a tag set to 0 or 2 after computing the source graph, the reporting neighbor is set to be the router which sent the update message. Together with Lemma 1, this implies that the Theorem is true. □

Theorem 4 *All the routers of a connected network have the most up-to-date link-state information needed to compute shortest paths to all destinations.*

Proof: The result is immediate from Theorem 2 in the absence of link failures. Consider the case in which the only link that fails in the network by time t_0 is link (s, d). Call this time $t_f \leq t_0$. According to ALP's operation, router s sends an LSU reporting an infinite cost for (s, d) within a finite time after t_f; furthermore, every router receiving the LSU reporting the infinite cost of (s, d) must forward the LSU if the link exists in its topology graph, i.e., the LSU gets flooded to all routers in the network that had heard about the link, and this occurs within a finite time after t_0. It than follows that no router in the network can use link (s, d) for any shortest path within a finite time after t_0. DELETE updates will also be propagated by router s for all those links in the topology graph that had router d as the reporting neighbor, as described in the proof of Lemma 2. A router sends and LSU for a link l to the router that transmitted a DELETE update if l is in the source graph and the router is not the reporting neighbor of l. Accordingly, within a finite time after t_0 all routers must only use links of finite cost in their source graphs; together with Theorem 2, this implies that the Theorem is true. □

Theorem 5 *If destination j becomes unreachable from a network component C at t_0; the topology graph of all routers in C includes no finite-length path to j.*

Proof: ALP's operation is such that, when a link fails, its head node reports an LSU with an infinite cost to its neighbors, and the state of a failed link is flooded through a connected component of the network together with DELETE updates for those links j that are part of the disconnected component to all those routers that knew about the link. Because a node failure equals the failure of all its adjacent links, it is true that no router in C can compute a finite-length path to j from its topology graph after a finite time after t_0. □

Note that, if a connected component remains disconnected from a destination j all link-state information corresponding to links for which j is the head node is updated when the network components get connected.

The previous theorems show that ALP sends correct routing tables within a finite time after link costs change, without the need to replicate topology information at every router (like OSPF does) or use explicit delete updates to delete obsolete information every time the source graph of a router changes (like LVA does).

2.4 Performance

ALP has the same communication, storage, and time complexity than LVA. However, worst-case performance is not truly indicative of ALP's performance advantage over LVA. Because link-states are deleted from the topology graph of a router, rather than after receiving explicit delete updates from neighbors, ALP incurs less communication overhead than LVA. ALP also compares favorably against recent distance vectors based on "source tracing" [6] [47], or the diffusion of distances [18], which do solve the looping problems of RIP and RIP-2.

Compared to the diffusion of distances, ALP disseminates link-state information from only the source of an LSU out to those routers that need the link, while DUAL requires distances to be disseminated from the source of the update out to those routers whose path included the source of the update, followed by replies going back to the source. Hence, when such coordination occurs in DUAL, ALP incurs half the communication overhead.

Compared to source tracing algorithms, it is interesting to observe that in ALP a router notifies its routing tree to its neighbors by specifying each link in the tree, while in a source tracing algorithm the same tree is specified by reporting, for each node on the tree, the distance from the root of the tree to the node and the identification of the previous node on the tree. Clearly, there is an one-to-one mapping between the two representations, which means that the same routers will receive LSUs or distance-vectors updates reporting changes to the routing tree. In other words, the communication overhead is the same. Furthermore, in terms of communication overhead, it is not possible to attain a smaller overhead than sending updates (of links or distances) to only those routers whose shortest paths are affected by a topology change, i.e., ALP and source-tracing algorithms make very efficient use of communication, and both amount to a more distributed implementation of Dijkstra's SPF algorithm than protocols using topology broadcast (e.g., OSPF), which *replicate* SPF runs at each router.

In terms of storage overhead, ALP has similar overhead than distance-vector protocols and link-state protocols for the case of shortest-path routing. ALP and other link-state protocols become more attractive than distance-vector protocols when providing multiple paths to the same destinations becomes necessary.

Because of the way in which ALP updates link-state information, ALP outperforms any topology broadcast protocol. Because ALP does not use "delete" updates we expect ALP to outperform LVA, specially when nodes fail or resources recover. Furthermore, because no counting-to-infinity occurs in ALP, ALP should outperform protocols based on the Bellman-Ford algorithm. To verify this, we ran a number of simulation experiments to compare its average performance against DBF, topology-broadcast (called LSA in prior literature), and LVA. We used the same topology and experiment reported in [19] in order to compare ALP against the best-performing published results for other approaches. The performance metrics consist of the number of steps and update messages that are required for each algorithm to converge (i.e., the algorithm stops sending messages), and the size of these updates. When a router receives an update message, it compares its local step counter with the sender's counter, takes the maximum and increments the count. Update messages are processed one at a time in the order in which they arrive. Like LVA and LSA, ALP uses Dijkstra's algorithm to compute the local shortest-path tree. The results presented are based on simulations for the

DOE-ESNET topology (26-node wired network) [19] which was used in order to simply use published simulation results for the competing approaches. The graphs in Figure 2.7 show the results for every single link changing cost from 1 to 2; in Figures 2.8 and 2.9 for every link failing and recovering; as well as every node failing and recovering again (Figures 2.10 and 2.11). All changes were performed one at a time, and the algorithms had time to converge before the next change occurred. The ordinate of Figures 2.7, 2.8, and 2.9 represent identifiers of the links, and the ordinate of Figures 2.10 and 2.11 represent the identifiers of the nodes that are altered in the simulation.

ALP, DBF, and LVA propagate updates to only those routers affected by single link-cost changes (Figure 2.7). In contrast, LSA shows almost constant behavior because the same link-state update must be sent to all routers; ALP is the most efficient of the four algorithms. Each update message contains one link-state update in LSA, and an average of 1.10 links in ALP; the average number of messages transmitted in ALP is 43.36, 48.67 in LVA, and 57.45 in DBF.

Figure 2.8 depicts DBF suffering from *counting to infinity* in some cases. There is a small difference in the average number of updates and synchronization steps required in ALP and LVA. The average size of an ALP message is 2.40.

When a failed link recovers, ALP is superior to all three algorithms. The average number of messages in LVA is 70% more than in ALP; LSA exhibits the same behavior as with link-cost changes, and in average more than three times the number of update messages generated by ALP. With an average of 5.85 steps, ALP is twice as fast as LSA, and 50% faster than LVA. Messages in LSA are no longer one-link long due to the packets containing complete topology information sent over the recovering link.

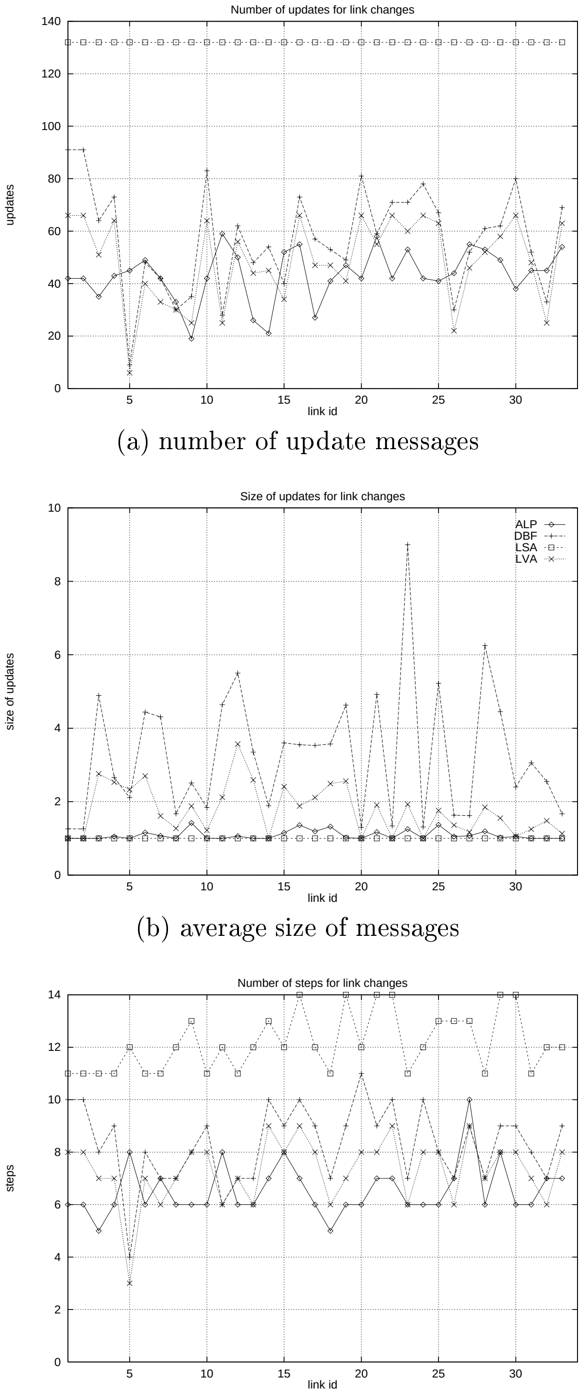
ALP also shows to have the best performance of the four algorithms for failing nodes.

DBF always suffers from *counting to infinity.* ALP needs to send 23% fewer updates than LSA, and 80% less the amount experienced by LVA. For recovering nodes, ALP shows to be more efficient than LVA, DBF, and LSA, both in terms of the amount of information sent through the network and speed of convergence.

The simulation results show that ALP has better overall average performance than LVA, LSA, and DBF. ALP behaves better than DBF and LVA when link cost changes and is always faster and produces less overhead traffic than LVA and LSA when resources are added to the network, and behaves better than the ideal LSA when links or routers fail. This is precisely the desired result, and indicates that ALP is desirable even if multiple constraints are not an issue.

2.5 Conclusions

We have presented, verified, and analyzed ALP. ALP is currently running in a small testbed implemented with PCs running gateD, and the very same code was used in the reported simulation experiment. The size of ALP's executable code, including the Hello Protocol and the Retransmission Protocol (Figure 2.2) is 96 Kbytes, compared to the 226 Kbytes of OSPF. Novel features in ALP include using three types of link state for any given link to disseminate correctly partial link-state information, and using designated routers per link for each broadcast medium.. Simulations using the actual gateD code for ALP corroborate the fact that ALP achieves the most efficient way of updating routing tables compared to topology broadcast, the distributed Bellman-Ford algorithm, and LVA. ALP addresses the complexity of today's approach to link-state routing by making the computation of routing trees using link-states costs a distributed computation.

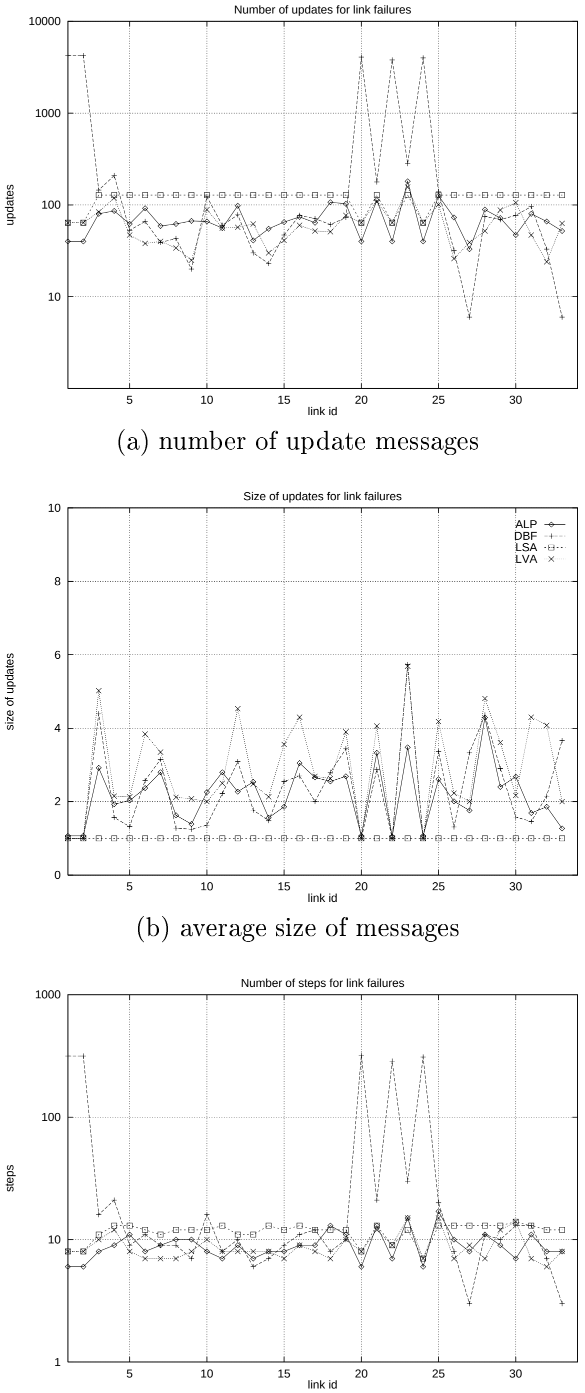


(a) number of update messages

(b) average size of messages

(c) number of steps for convergence

Figure 2.7: Results for links changing cost

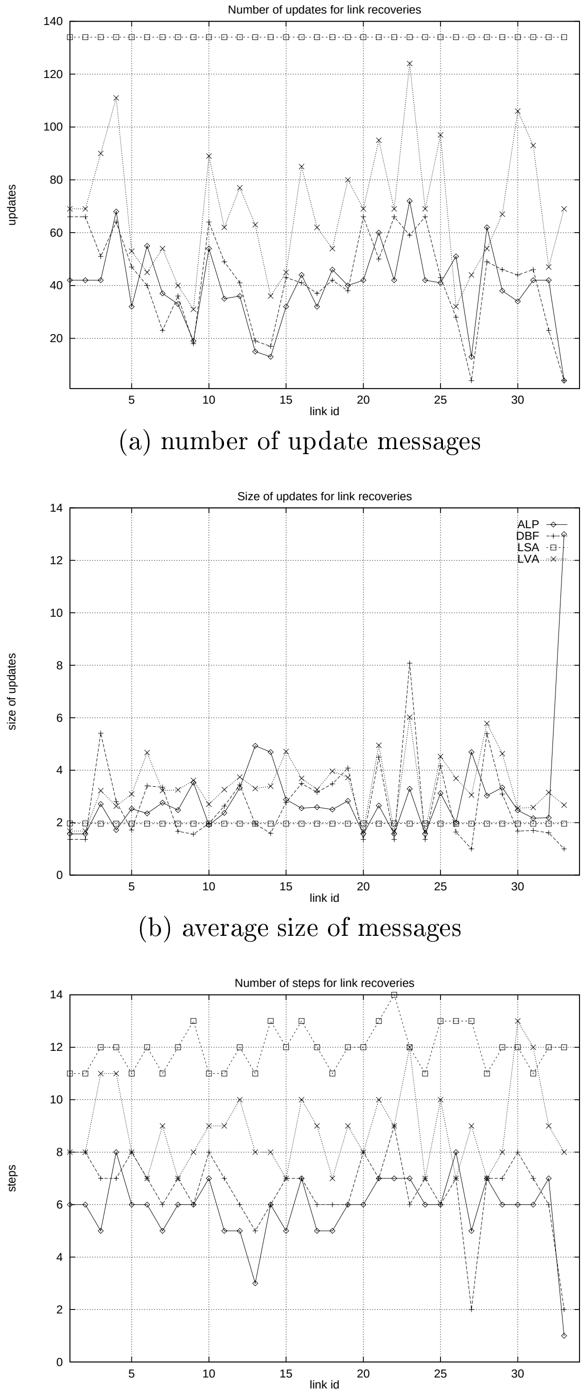


(a) number of update messages

(b) average size of messages

(c) number of steps for convergence

Figure 2.8: Results for links failing

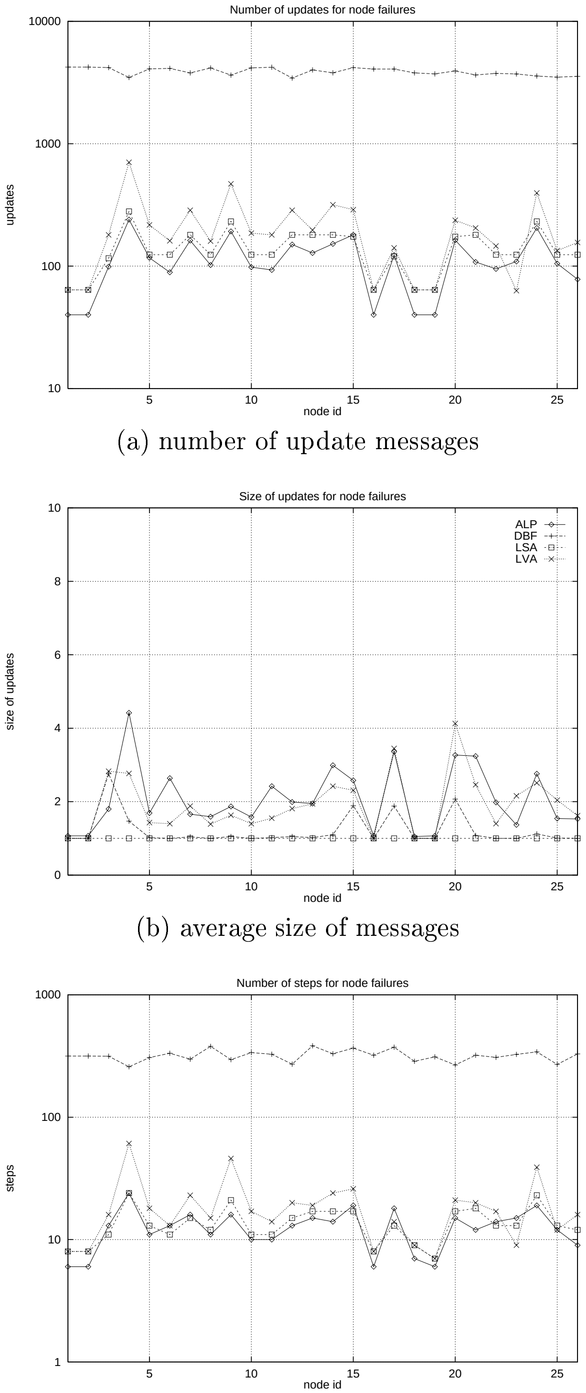


(a) number of update messages

(b) average size of messages

(c) number of steps for convergence

Figure 2.9: Results for links recovering after failure

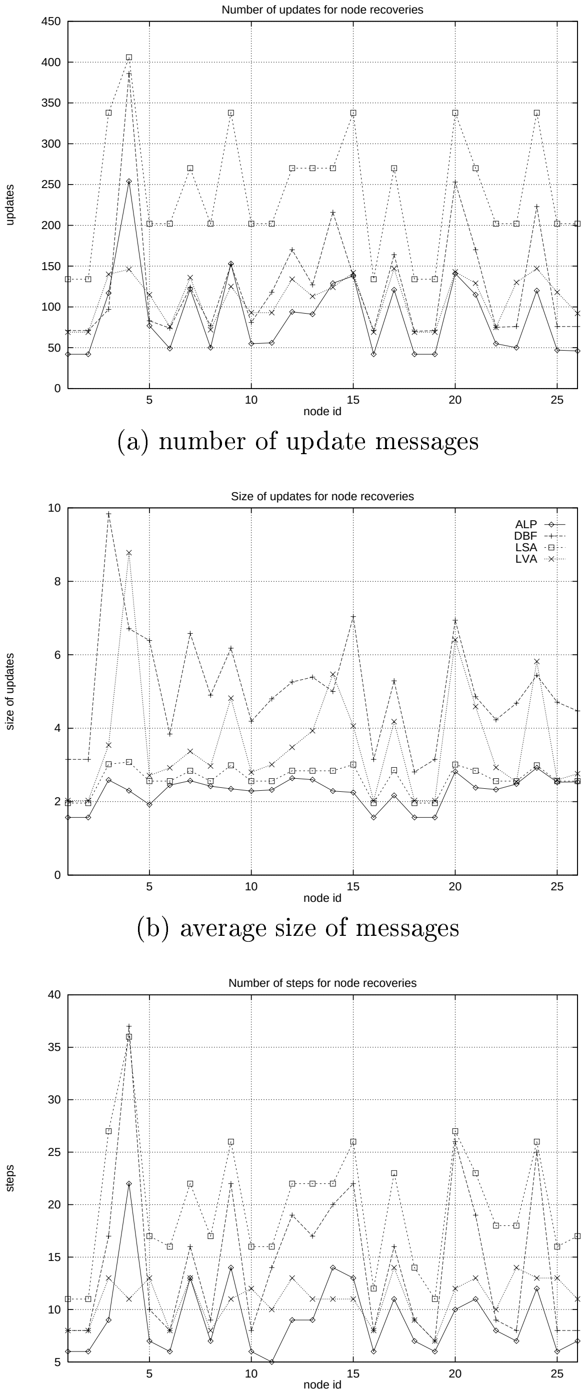


(a) number of update messages

(b) average size of messages

(c) number of steps for convergence

Figure 2.10: Results for nodes failing



(a) number of update messages

(b) average size of messages

(c) number of steps for convergence

Figure 2.11: Results for nodes recovering after failure

Chapter 3

Source Tree Routing

The intuition behind the approach used in the source-tree adaptive routing (STAR) protocol can be stated as follows. In an on-demand routing protocol, every source polls all the destinations to find paths to a given destination; conversely, in a table-driven routing protocol, every destination polls all the sources in the sense that they obtain paths to a destination resulting from updates originated by the destination. Therefore, given that some form of flooding occurs in either approach, it should be possible to obtain a table-driven protocol that needs to poll as infrequently as on-demand routing protocols do to limit the overhead of the routing protocol.

In STAR, a router sends updates to its neighbors regarding the links in its preferred paths to destinations. The links along the preferred paths from a source to each desired destination constitute a *source tree* that implicitly specifies the complete paths from the source to each destination. Each router computes its source tree based on information about adjacent links and the source trees reported by its neighbors, and reports changes to its source tree to all its neighbors incrementally. The aggregation of adjacent links and source trees reported by neighbors constitutes the partial topology known by a router. STAR can be used with distributed hierarchical routing schemes proposed in the past for both distance-vector or link-state routing [32, 10, 37, 2].

Prior proposals for link-state routing using partial link-state data without clusters [19, 20] require routers to explicitly inform their neighbors which links they use and which links they stop using. In contrast, because STAR sends only changes to the structure of source trees, and because each destination has a single predecessor in a source tree, a router needs to send only updates for those links that are part of the tree and a single update entry for the root of any subtree of the source tree that becomes unreachable. Routers receiving a STAR update can infer correctly all the links that the sender has stopped using, without the need for explicit delete updates.

Section 3.1 describes two different approaches that can be used to update routing information in wireless networks: the optimum routing approach (ORA) and the least-overhead routing approach (LORA), and elicit the reasons why STAR is the first table-driven routing protocol that can adopt LORA. Section 3.2 describes STAR and how it supports ORA and LORA. Section 3.3 demonstrates that routers executing STAR stop disseminating link-state updates and obtain shortest paths to destinations within a finite time after the cost of one or more links changes. Section 3.4 compares STAR's performance against the performance of other table-driven and on-demand routing protocols using simulation experiments. These experiments use the same methodology described by Broch et al. to compare on-demand routing protocols [11], and our simulation code is the same code that runs in embedded wireless routers; the code we used for DSR was ported from the ns2 code for DSR available from CMU [45]. The simulation results show that STAR is four times more bandwidth-efficient than the best-performing link-state routing protocol previously proposed, an order of magnitude more bandwidth-efficient than topology broadcasting, and, depending on the scenario, 1.3 to 6 times more bandwidth-efficient than DSR, which has been shown to incur the least overhead among several on-demand routing protocols [11]. To our knowledge, this is the first time that any table-driven routing protocol has been shown to be more efficient than on-demand routing protocols in wireless networks.

3.1 Updating Routes in Wireless Networks

We can distinguish between two main approaches to updating routing information in the routing protocols that have been designed for wireless networks and that are based on nodal identifiers rather than node coordinates: the *optimum routing approach* (ORA) and the *least-overhead routing approach* (LORA). With ORA, the routing protocol attempts to update routing tables as quickly as possible to provide paths that are optimum with respect to a defined metric. In contrast, with LORA, the routing protocol attempts to provide viable paths according to a given performance metric, which need not be optimum, to incur the least amount of control traffic.

For the case of ORA, the routing protocol can provide paths that are optimum with respect to different types of service (TOS), such as minimum delay, maximum bandwidth, least amount of interference, maximum available battery life, or combinations of metrics. Multiple TOS can be supported in a routing protocol; however, this paper focuses on a *single* TOS to address the performance of routing protocols providing ORA, and uses shortest-path routing as the single TOS supported for ORA. We assume that a single metric, which can be a combination of parameters, is used to assign costs to links.

On-demand routing protocols such as DSR follow LORA, in that these protocols attempt to minimize control overhead by: (a) maintaining path information for only those destinations with which the router needs to communicate, and (b) using the paths found after a flood search as long as the paths are valid, even if the paths are not optimum. On-demand routing protocols can be applied to support multiple TOS; an obvious approach is to obtain paths of different TOS using separate flood searches. However, we assume that a single TOS is used in the network. ORA is not an attractive or even feasible approach in on-demand routing protocols, because flooding the network frequently while trying to optimize existing paths with respect to a cost metric of choice consumes the available bandwidth and can make the paths worse while trying to optimize them.

We can view the flood search messages used in on-demand routing protocols as a form of polling of destinations by the sources. In contrast, in a table-driven routing protocol, it is the destinations who poll the sources, meaning that the sources obtain their paths to destinations as a result of update messages that first originate at the destinations. What is apparent is that some form of information flooding occurs in both approaches.

Interestingly, all the table-driven routing protocols reported to date for ad hoc networks adhere to ORA, and admittedly have been adaptations of routing protocols developed for wired networks. A consequence of adopting ORA in table-driven routing within a wireless network is that, if the topology of the network changes very frequently, the rate of update messages increases dramatically, consuming the bandwidth needed for user data. The two methods used to reduce the update rate in table-driven routing protocols are clustering and sending updates periodically. Clustering is attractive to reduce overhead due to network size; however, if the affiliations of nodes with clusters change too often, then clustering itself introduces unwanted overhead. Sending periodic updates after long timeouts reduces overhead, and it is a technique that has been used since the DARPA packet-radio network was designed [31]; however, control traffic still has to flow periodically to update routing tables.

A nice feature of such routing protocols as DSR [30] and WIRP [12] is that these protocols remain quiet when no new update information has to be exchanged; they have no need for periodic updates. Both protocols take advantage of promiscuous listening of packets sent by neighbor routers to determine the neighborhood of the router. A key difference between DSR and WIRP is that DSR follows LORA while WIRP follows ORA, which means that WIRP may incur unnecessary overhead when the network topology is unstable.

Given that both on-demand and table-driven routing protocols incur flooding of information in one way or another, a table-driven routing protocol could be designed that incurs similar or less overhead than on-demand routing protocols by limiting the polling done by the destinations to be the same or less than the polling done by the sources in on-demand routing protocols. However, there has been no prior description of a table-driven routing protocol that can truly adhere to LORA, i.e., one that uses node identifiers for routing, has no need for periodic updates, uses no clustering, and remains quiet as long as the paths available at the routers are valid, even if they are not optimum. The reason why no prior table-driven routing protocols have been reported based on LORA is that, with the exception of WIRP and WRP, prior protocols have used either distances to destinations, topology maps, or subsets of the topology, to obtain paths to destinations, and none of these types of information permits a router to discern whether the paths it uses are in conflict with the paths used by its neighbors. Accordingly, routers must send updates after they change their routing tables in order to avoid long-term routing loops, and the best that can be done is to reduce the control traffic by sending such updates periodically. In the next section, we describe STAR, which is the first table-driven routing protocol that implements LORA.

3.2 STAR Description

3.2.1 Network Model

In STAR, routers maintain a partial topology map of their network. In this paper we focus on flat topologies only, i.e., there is no aggregation of topology information into areas or clusters.

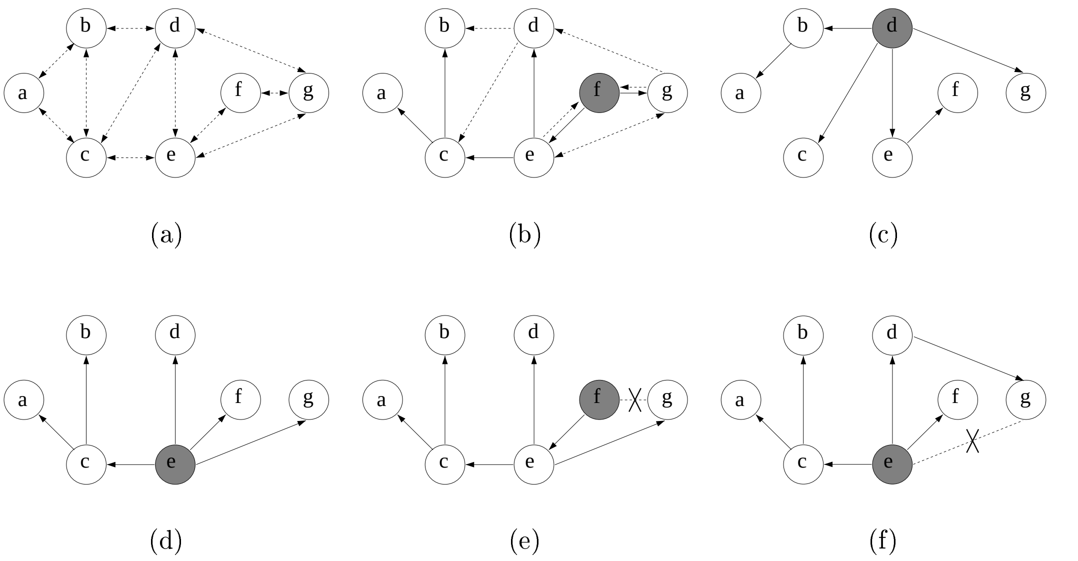
To describe STAR, the topology of a network is modeled as a directed graph $G = (V, E)$, where V is the set of nodes and E is the set of edges connecting the nodes. Each node has a unique identifier and represents a router with input and output queues of unlimited capacity updated according to a FIFO policy. In a wireless network, a node can have connectivity with multiple nodes in a single physical radio link. For the purpose of routing-table updating, a node A can consider another node B to be adjacent (we call such a node a "neighbor") if there is link-level connectivity between A and B and A receives update messages from B reliably. Accordingly, we map a physical broadcast link connecting multiple nodes into multiple point-to-point bidirectional links defined for these nodes. A functional bidirectional link between two nodes is represented by a pair of edges, one in each direction and with a cost associated that can vary in time but is always positive.

In our description of STAR, we assume that an underlying protocol assures that a router detects within a finite time the existence of a new neighbor and the loss of connectivity with a neighbor. However, depending on the MAC protocol used in the ad hoc network, determining who the neighbors of a node are may be done based on promiscuous listening of packets transmitted by neighbors. All messages, changes in the cost of a link, link failures, and new-neighbor notifications are processed one at a time within a finite time and in the order in which they are detected. Routers are assumed to operate correctly, and information is assumed to be stored without errors.

3.2.2 Overview

In STAR, each router reports to its neighbors the characteristics of every link it uses to reach a destination. The set of links used by a router in its preferred path to destinations is called the *source tree* of the router. A router knows its adjacent links and the source trees reported by its neighbors; the aggregation of a router's adjacent links and the source trees reported by its neighbors constitute a partial *topology graph*. The links in the source tree and topology graph must be adjacent links or links reported by at least one neighbor. The router uses the topology graph to generate its own source tree. Each router derives a routing table specifying the successor to each destination by running a local *route-selection algorithm* on its source tree. A critical aspect of the route selection algorithm is that a router can choose a neighbor as its successor to a destination only if that neighbor has reported having a source tree containing a path to the destination that does not involve the router itself.

Under LORA, a router running STAR sends updates on its source tree to its neighbors only when it loses all paths to one ore more destinations, when it detects a new destination, or when it determines that local changes to its source tree can potentially create long term routing loops. Because each router communicates its source tree to its neighbors, the deletion of a link no longer used to reach a destination is implicit with the addition of the new link used to reach the destination and need not be sent explicitly as an update; a router makes explicit reference to a failed link only when the deletion of a link causes the router to have no paths to one or more destinations, in which case the router cannot provide new links to make the deletion of the failed link implicit. The example shown in Figure 3.1 illustrates how link failures may not cause the generation of update messages when STAR is running under LORA. All links and nodes are assumed to have the same propagation delays, and all the links have unit cost. Figures 3.1(b) through 3.1(d) show the source trees according to STAR



(a) (b) (c)

(d) (e) (f)

Figure 3.1: An example topology

at the routers indicated with filled circles for the network topology depicted in Figure 3.1(a), solid lines represent the links that are part of the source tree of the nodes, and the dashed lines in Figure 3.1(b) represent the links that are in the topology graph of router f but not in f's source tree. After processing the failure of link (f, g) (Figure 3.1(e)) router f does not need to report the changes to its source tree because all the destinations are reachable and no permanent routing loop can be formed. The same is true when node e processes the failure of link (e, g).

The basic update unit used in STAR to communicate changes to source trees is the link-state update (LSU). An LSU reports the characteristics of a link; an update message contains one or more LSUs. For a link between router u and router or destination v, router u is called the *head node* of the link in the direction from u to v. The head node of a link is the only router that can report changes in the parameters of that link. LSUs are validated using time stamps, and each router erases a link from its topology graph if the link is not present in the source trees of any of its neighbors.

3.2.3 Information Stored and Exchanged

We assume in the rest of the paper that a single parameter is used to characterize a link in one of its directions, which we will call the cost of the directed link. Furthermore, although any type of local route selection algorithm can be used in STAR, we describe STAR assuming that Dijkstra's shortest-path first (SPF) algorithm is used at each router to compute preferred paths.

An LSU for a link (u, v) in an update message is a tuple (u, v, l, t) reporting the characteristics of the link, where l represents the cost of the link and t is the time stamp assigned to the LSU.

A router i maintains a topology graph TG_i, a source tree ST_i, a routing table, the set of neighbors N_i, the source trees ST_x^i reported by each neighbor $x \in N_i$, and the topology graphs TG_x^i for each neighbor $x \in N_i$. The topology graph TG_x^i contains the links in ST_x^i and the links reported by neighbor x in a message being processed by router i, after processing the message $TG_x^i \equiv ST_x^i$.

The record entry for a link (u, v) in the topology graph of router i is denoted $TG_i(u, v)$ and is defined by the tuple (u, v, l, t, del), and an attribute p in the tuple is denoted by $TG_i(u, v).p$. The same notation applies to a link (u, v) in ST_i, ST_x^i, and TG_x^i. $TG_i(u, v).del$ is set to TRUE if the link is not in the source tree of any neighbor.

A vertex v in TG_i is denoted $TG_i(v)$. It contains a tuple $(d, pred, suc, d', d'', suc', nbr)$ whose values are used on the computation of the source tree. $TG_i(v).d$ reports the distance of the path $i \rightsquigarrow v$, $TG_i(v).pred$ is v's predecessor in $i \rightsquigarrow v$, $TG_i(v).suc$ is the next hop along the path towards v, suc' holds the address of the previous hop towards v, d' corresponds to the previous distance to v reported by suc', d'' is the cost of the path $i \leadsto v$ the last time the cost of the path changed by Δ, and nbr is a flag used to determine if an update message must be generated when the distance reported by the new successor towards v increases. The same notation applies to a vertex v in ST_i, ST_x^i, and TG_x^i.

The source tree ST_i is a subset of TG_i. The routing table contains record entries for destinations in ST_i, each entry consists of the destination address, the cost of the path to the destination, and the address of the next-hop towards the destination.

A router i running LORA also maintains the last reported source tree $ST_i{}'$.

The cost of a failed link is considered to be infinity. The way in which costs are assigned to links is beyond the scope of this specification. As an example, the cost of a link could simply be the number of hops, or the addition of the latency over the link plus some constant bias.

We refer to an LSU that has an infinite cost as a RESET; furthermore $TG_i^i \equiv TG_i$, and $ST_i^i \equiv ST_i$.

3.2.4 Validating Updates

Because of delays in the routers and links of an ad hoc network, update messages sent by a router may propagate at different speeds along different paths. Therefore, a given router may receive an LSU from a neighbor with stale link-state information, and a distributed termination-detection mechanism is necessary for a router to ascertain when a given LSU is valid and avoid the possibility of LSUs circulating forever. STAR uses time stamps to validate LSUs. A router either maintains a clock that does not reset when the router stops operating, or asks its neighbors for the oldest known time stamp after it initializes or reboots.

A router receiving an LSU accepts the LSU as valid if the received LSU has a larger time stamp than the time stamp of the LSU stored from the same source, or if there is no entry for the link in the topology graph and the LSU is not reporting an infinite cost. Link-state information for failed links is erased from the topology graph after it ages out, which takes the order of an hour after having processed the LSU of a link. LSUs for operational links are erased from the topology graph when the links are erased from the source tree of all the neighbors.

Routers running STAR need to keep the state of failed links in their topology graphs until all the nodes in a connected network are aware that there exist no path to a node that has failed or has lost connectivity to all its neighbors. If a node that had more than one neighbor fails and the nodes in the network delete the state of failed links from the topology graph, then the termination-detection mechanism fails in those nodes that knew about the state of more than one link which had the failed node as the tail of the link. Consequently, some nodes in the network may keep a route to the failed destination, not necessarily creating routing loops, wasting bandwidth by forwarding data packets to the unreachable destination.

When STAR is running under LORA the head node of a failed link only reports to its neighbors the failure of the link if the tail node of the link becomes unreachable, i.e., all the nodes that have the failed link in their source trees will be unaware of the new state of the link. Consequently, the head node of the failed link may receive an outdated LSU for the failed link in the first update message transmitted by a new neighbor. For STAR to work properly under LORA, whenever a node receives an LSU for one of its outgoing links and the link is not present in its topology graph or the link has an infinite cost, the node must add an entry (if there is none) for the link in its topology graph with an infinite cost and (re-)start aging the link. A failed link can be deleted from a router's topology graph only if the link is not present in the reported source tree of any neighbor.

We note that, because LSUs for operational links never age out, there is no need for the head node of a link to send periodic LSUs to update the time stamp of the link. This means that STAR disseminates LSUs for a given link only when changes are made to the source tree affecting the link. This is in contrast to other routing protocols based on sequence numbers or time stamps together with aging, which age out LSUs and must, therefore, flood an LSU for a given link within a fixed time interval in the absence of changes to the link.

3.2.5 Exchanging Update Messages

How update messages are exchanged depends on the routing approach used (ORA or LORA). The rest of this section describes how LORA and ORA can be supported in STAR.

For ORA to be supported in STAR, the only rule needed for sending update messages consists of a router sending an update message every time its source tree changes.

In an on-demand routing protocol, a router can keep using a path found as long as the path leads to the destination, even if the path does not have optimum cost. A similar approach can be used in STAR, because each router has a complete path to every destination as part of its source tree. To support LORA, router i running STAR reports updates to its source trees in the event of unreachable destinations, new destinations, the possibility of permanent routing loops, and cost of paths exceeding a given threshold. Router i accomplishes this by comparing its source tree against the source trees it has received from its neighbors after any input event, and by sending the updates to its source tree according to the following three rules.

LORA-1: Router i sends a source-tree update when it finds a new destination, or any of its neighbors reports a new destination.

Whenever a router hears from a new neighbor that is also a new destination, it sends an update message that includes the new LSUs in its source tree. Obviously, when a router is first initialized or after a reboot, the router itself is a new destination and should send an update message to its neighbors. Link-level support should be used for the router to know its neighbors within a short time, and then report its links to those neighbors with LSUs sent in an update message. Else, a simple way to implement an initialization action consists of requiring the router to listen for some time for neighbor traffic, so that it can detect the existence of links to neighbors.

LORA-2: Router i sends a source-tree update when the change in the cost of the path to at least one destination exceeds a threshold Δ for router i or any of its neighbors.

In this paper, we assume $\Delta = \infty$, i.e., routers force source-tree updates when destinations become unreachable. When a router processes an input event (e.g., a link fails, an update message is received) that causes *all* its paths through all its neighbors to one or more destination to be severed, the router sends an update message that includes an LSU specifying an infinite cost for the link connecting to the head of each subtree of the source tree that becomes unreachable. The update message does not have to include an LSU for each node in an unreachable subtree, because a neighbor receiving the update message has the sending node's source tree and can therefore infer that all nodes below the root of the subtree are also unreachable, unless LSUs are sent for new links used to reach some of the nodes in the subtree. When at least one destination becomes unreachable to any of the router's neighbors and the router has a path to that destination then the router sends an update message reporting the changes to its source tree.

LORA-3: Router i sends a source-tree update after processing an input event if:

1. A path implied in the source tree of router i leads to a loop.
2. The new successor chosen to a given destination has an address greater than the address of router i.

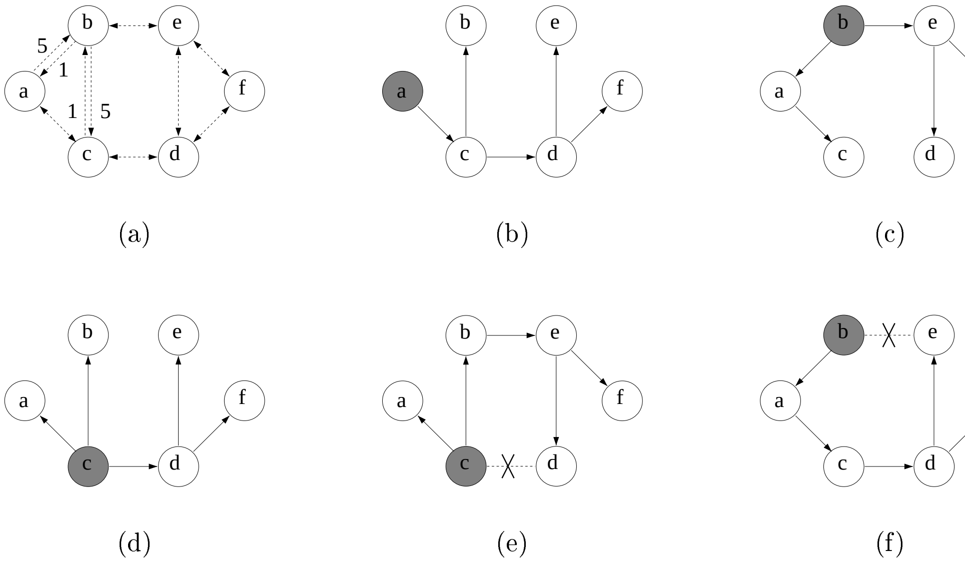
3. The reported distance from the new chosen successor n to a destination j is longer than the reported distance from the previous successor to the same destination. However, if the cost of the path $i \rightsquigarrow j$ increases and $n \neq i$ is a neighbor of j, no update message is needed regarding j or any destination whose path from i involves j.

The loop-prevention mechanisms of LORA-3 assume that the local route selection algorithm is such that a router cannot add a link (u, v) to its new source tree choosing neighbor k as the successor to v if (u, v) is not in the source tree reported by k. This is easily done by labeling each link in the topology graph with the neighbors that have reported the link, and by allowing a link (u, v) to be added to the new source tree only if the neighbor k used in the path from the root of the source tree to (u, v) is one of the reporting neighbors of the link.

To explain the need for the first part of LORA-3, we observe that, in any routing loop among routers with unique addresses, one of the routers must have the smallest address in the loop; therefore, if a router is forced to send an update message when it chooses a successor whose address is greater than its own, then it is not possible for all routers in a routing loop to remain quiet after choosing one another, because at least one of them is forced to send an update message, which causes the loop to break when routers update their source trees.

The last part of LORA-3 is needed when link costs can assume different values in different directions, in which case the first part of LORA-3 may not suffice to break loops because the node with the smallest address in the loop may not have to change successors when the loop is formed. The following example illustrates this scenario.

Consider the six-node wireless network shown in Figure 3.2 and assume that the last part of LORA-3 is not in effect at the routers running STAR. In this example, nodes are given identifiers that are lexicographically ordered, i.e., a is the smallest identifier and f is the largest



(a) (b) (c)

(d) (e) (f)

Figure 3.2: Routers running STAR without the last part of LORA-3 being in effect.

identifier in the graph. All links and nodes are assumed to have the same propagation delays, and all the links but links (a, b) and (b, c) have unit cost. Figures 3.2(b) through 3.2(d) show the source trees according to STAR at the routers indicated with filled circles for the network topology depicted in Figure 3.2(a). Arrowheads on solid lines indicate the direction of the links stored in the router's source tree. Figure 3.2(e) shows c's new source tree after processing the failure of link (c, d); we note that c does not generate an update message, because $c > b$ by assumption. Suppose link (b, e) fails immediately after the failure of (c, d), node b computes its new source tree shown in Figure 3.2(f) without reporting changes to it because a is its new successor to destinations d, e, and f, and $a < b$. A permanent loop forms among nodes a, b, and c.

Figure 3.3 depicts the sequence of events triggered by the execution of the last part of LORA-3 in the same example introduced in Figure 3.2, after the failures of links (c, d) and (b, e). The figure shows the LSUs generated by the node with filled circle transmitted in an

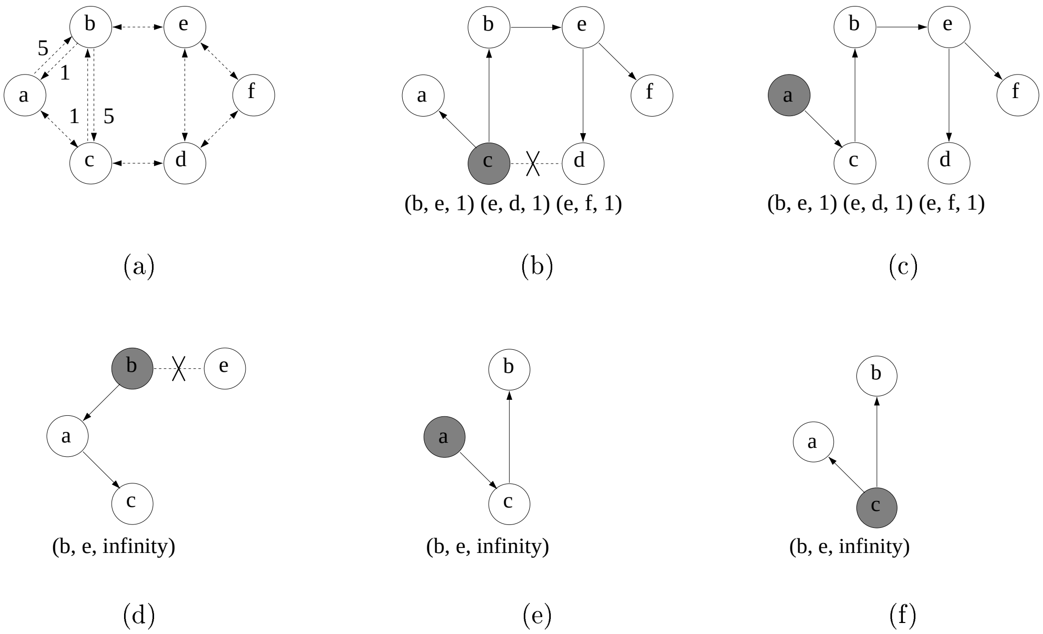


Figure 3.3: Routers running STAR with the last part of LORA-3 being in effect.

update message to the neighbors, and shows such LSUs in parentheses. The third element in an LSU corresponds to the cost of the link (a RESET has cost $infinity$). Unlike in the previous example, node c transmits an update message after processing the failure of link (c, d) because of the last part of LORA-3; the distance from the new successor b to d and f is greater than from the previous successor d. When link (b, e) fails, node b realizes that the destinations d, e, and f are unreachable and generates an update message reporting the failure of the link connecting to the head of the subtree of the source tree that becomes unreachable. The update message from b triggers the update messages that allow nodes a, b, and c to realize that there are no paths to d, e, and f. A similar sequence of events takes place at the other side of the network partition.

The example shown in Figure 3.4 illustrates the scenario in which a router that chooses a new successor to a destination with a larger distance to it does not need to send an update

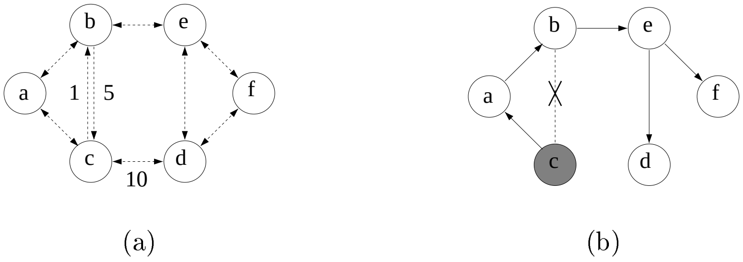


Figure 3.4: The last part of LORA-3 not always triggers the generation of an update message: (a) network topology, and (b) source tree of node c after processing the failure of link (c, b).

message. Figure 3.4(b) shows the new source tree of node c after the failure of link (c, b). In this case, c does not need to send an update message because the parent node of the subtree headed by b is a neighbor of c and therefore no permanent loop can be formed.

To ensure that the above rules work with incremental updates specifying only changes to a source tree, a router must remember the source tree that was last notified to its neighbors. If any of LORA-1 to LORA-3 are satisfied, the router must do one of two things:

- If the new source tree includes new neighbors than those present in the source tree that was last updated, then the router must send its entire source tree in its update, so that new neighbors learn about all the destinations the router knows.
- If the two source trees imply the same neighbors, the router sends only the updates needed to obtain the new tree from the old one.

The above rules are sufficient to ensure that every router obtains loopless paths to all known destinations, without the routers having to send updates periodically. In addition to the ability for a router to prevent loops in STAR, the two key features that enable STAR to adopt LORA are: (a) validating LSUs without the need of periodic updates, and (b) the ability to either listen to neighbors' packets or use a neighbor protocol at the link layer to determine who the neighbors of a router are.

The rules for update-message exchange stated above assume that an update message is sent reliably to all the neighbors of a router. As the performance analysis of Section 3.4 shows, this is a very realistic assumption, because STAR working under LORA generates far fewer update messages than the topology changes that occur in the network. However, if preserving bandwidth is of utmost importance and the underlying link protocol is contention-based, additional provisions must be taken as described in the next section.

3.2.6 Impact of The Link Layer in LORA

The rules for update-message exchange stated in the previous section assume that an update message is sent reliably to all the neighbors of a router. If the link layer provides efficient reliable broadcast of network-level packets, then STAR can rely on sending an update message only once to all neighbors, with the update message specifying only incremental changes to the router's source tree. The link layer will retransmit the packet as needed to reach all neighbors, so that it can guarantee that a neighbor receives the packet unless the link is broke. An alternative way to provide a reliable exchange of update messages consists of providing collision-free broadcasts of update messages at the medium access control (MAC) layer and implementing the retransmission strategy for update messages as part of STAR itself.

A reliable broadcast service at the link layer can be implemented very efficiently at the link layer or in STAR itself if the MAC protocol used guarantees collision-free transmissions of broadcast packets. A typical example of A MAC protocol that can support collision-free broadcasts is TDMA, and there are several recent proposals that need not rely on static assignments of resources (e.g., FPRP [56], CARTS [54]).

Unfortunately, reliable broadcasting from a node to all its neighbors is not supported in the collision-avoidance MAC protocols that have been proposed [16, 29, 7] or implemented in commercial products for ad hoc networks operating in ISM bands. Furthermore, any link-level or network-level strategy for reliable exchange of broadcast update messages over a contention-based MAC protocol will require substantial retransmissions under high-load conditions and rapid changes to the connectivity of nodes. Therefore, if the underlying MAC protocol does not provide collision-free broadcasts, then STAR (and any table-driven routing protocol for that matter) is better off relying on the approach adopted in the past in the DARPA packet-radio network, whereby a router broadcasts unreliably its update messages to its neighbors, and each update message contains the entire source tree. For STAR to operate correctly with this approach under LORA, routers must prevent the case in which permanent loops are created because an update message is not received by a neighbor due to channel errors or hidden-terminal interference.

When the routers transmit updates over a MAC protocol that does not provide collision-free broadcasting, the following additional mechanisms are needed in STAR: (a) the data packets must record the route traversed, and (b) four additional rules are used to send an update messages. These added rules are used to provide persistence in the exchange of updates, probe neighbor routers for updates when paths to a destination are not known, and break loops detected by the traversal of data packets.

LORA-4: Router i sends its update message as a reliable unicast to the neighbor that makes router i send its update, and all neighbors of i process the update message.

LORA-5: A router sends periodic updates in intervals of 60 seconds while at least one of its neighbors does not report having a path to a destination known to the router; otherwise, periodic updates are transmitted in intervals of 600 seconds or longer.

LORA-6: When router i has a data packet to send to a destination j for which it has no paths, it sends an update message to its neighbors reporting the absence of a path to

j. This message acts as a query, because any neighbor with a path to j receiving the message will generate an update message and send it reliably to router i. While router i has no path to j, it retransmits its update message in intervals of 600 milliseconds, 6 seconds, and 60 seconds, and then backs off to periodic updates transmitted in intervals of 600 seconds or longer.

LORA-7: Router i receives a data packet to destination j and one of the routers in the traversed path is in i's path to the destination, the data packet is discarded and a ROUTE-REPAIR update message is generated to break the loop. A ROUTE-REPAIR contains the complete source tree of the sender's router and the *route repair path*, and is transmitted reliably to the router in the head of the route repair path. The *route repair path* corresponds to the path $i \rightsquigarrow x$, where x is the last router in the data packet's traversed path that is first found in the path $i \rightsquigarrow j \in ST_i$. When a router receives a ROUTE-REPAIR update it removes itself from the route repair path and transmits a ROUTE-REPAIR with its source tree to the head of the route repair path. When a router detects a loop it will only transmit a ROUTE-REPAIR update to neighbor k if 30 seconds have elapsed since the last time a ROUTE-REPAIR was sent to k.

3.2.7 Details on The Processing of Input Events

Figures 3.5 and 3.6 specify the main procedures of STAR (for both LORA and ORA approaches) used to update the routing table and the link-state database at a router i. It is assumed that the link layer provides reliable broadcast of network-level packets. Procedure *NodeUp* is executed when a router i starts up. The neighbor set of the router is empty initially.

If the neighbor protocol reports a new link to a neighbor k (procedure *NeighborUp*), the router then runs *Update* with the appropriate message as input; the LSU in the message gets

```
NodeUp()
description
  Node $i$ initializes itself
{
  $TG_i \leftarrow \emptyset$;
  $ST_i \leftarrow \emptyset$;
  $ST'_i \leftarrow \emptyset$;
  $N_i \leftarrow \emptyset$;
  $M_i \leftarrow$ FALSE;
  $NS_i \leftarrow$ FALSE;
}

NeighborUp($k$)
description
  Neighbor protocol reports connectivity
  to neighbor $k$
{
  $N_i \leftarrow N_i \cup \{k\}$;
  $TG^i_k \leftarrow \emptyset$;
  $ST^i_k \leftarrow \emptyset$;
  $sendST \leftarrow$ TRUE;

  if ( LORA and $k \in TG_i$ and $TG_i(k).pred \neq null$ )
  {
    $NS_i \leftarrow$ TRUE;
    $sendST \leftarrow$ FALSE;
  }

  Update($i$, $(i, k, l^i_k, T_i)$);

  if ( $sendST$ )
  {
    $MSG_i \leftarrow \emptyset$;

    for each ( link $(u, v) \in ST_i$ )
      $MSG_i \leftarrow MSG_i \cup \{( u, v, TG_i(u, v).l,$
                                 $TG_i(u, v).t)\}$;
  }

  Send();
}

NeighborDown($k$)
description
  Neighbor protocol reports link
  failure to neighbor $k$
{
  $N_i \leftarrow N_i - \{k\}$;
  $TG^i_k \leftarrow \emptyset$;
  $ST^i_k \leftarrow \emptyset$;

  Update($i$, $(i, k, \infty, T_i)$);

  Send();
}

LinkCostChange($k$)
description
  Neighbor protocol reports link
  cost change to neighbor $k$
{
  Update($i$, $(i, k, l^i_k, T_i)$);

  Send();
}

Update($k$, $msg$)
description
  Process update message $msg$
  sent by router $k$
{
  UpdateTopologyGraph($k$, $msg$);

  if ( $k \neq i$ )
    BuildShortestPathTree($k$, $null$);

  BuildShortestPathTree($i$, $k$);
  UpdateRoutingTable();

  if ( $k \neq i$ )
    Send();
}
```

```
UpdateTopologyGraph($k$, $msg$)
description
  Update $TG_i$ and $TG^i_k$ from LSUs in $msg$
{
  for each ( LSU $(u, v, l, t) \in msg$ )
  {
    if ( $l \neq \infty$ )
      ProcessAddUpdate($k$, $(u, v, l, t)$);
    else
      ProcessVoidUpdate($k$, $(u, v, l, t)$);
  }
}

ProcessAddUpdate($k$, $(u, v, l, t)$)
description
  Update topology graphs $TG_i$ and $TG^i_k$ from LSU
{
  if ( $(u, v) \notin TG_i$ or $t > TG_i(u, v).t$ )
  {
    if ( $(u, v) \notin TG_i$ )
    {
      $TG_i \leftarrow TG_i \cup \{(u, v, l, t)\}$;
      if ( LORA and $k \neq i$ and $u = i$ )
        $TG_i(u, v).l \leftarrow \infty$;
    }
    else
    {
      $TG_i(u, v).l \leftarrow l$;    $TG_i(u, v).t \leftarrow t$;
    }
  }
  if ( $k \neq i$ )
  {
    if ( $\exists\ (r, s) \in TG^i_k \mid r \neq u$ and $s = v$ )
      $TG^i_k \leftarrow TG^i_k - \{(r, s)\}$;
    if ( $(u, v) \notin TG^i_k$ )
      $TG^i_k \leftarrow TG^i_k \cup \{(u, v, l, t)\}$;
    else
    {
      $TG^i_k(u, v).l \leftarrow l$;    $TG^i_k(u, v).t \leftarrow t$;
    }
  }
  $TG_i(u, v).del \leftarrow$ FALSE;
  if ( $TG_i(u, v).l = \infty$ ) Start aging $(u, v)$;
}

ProcessVoidUpdate($k$, $(u, v, l, t)$)
description
  Update topology graphs $TG_i$ and $TG^i_k$ from LSU
{
  if ( $(u, v) \in TG_i$ )
  {
    if ( $t > TG_i(u, v).t$ )
    {
      $TG_i(u, v).l \leftarrow l$;    $TG_i(u, v).t \leftarrow t$;
      Start aging $(u, v)$;
    }
    if ( $k \neq i$ and $(u, v) \in TG^i_k$ )
    {
      $TG^i_k(u, v).l \leftarrow l$;    $TG^i_k(u, v).t \leftarrow t$;
    }
    $TG_i(u, v).del \leftarrow$ FALSE;
  }
}

Send()
{
  if ( $MSG_i \neq \emptyset$ ) Broadcast message $MSG_i$;
  $MSG_i \leftarrow \emptyset$;
}

InitializeSingleSource($k$)
{
  for each ( vertex $v \in TG^i_k$ )
  {
    $TG^i_k(v).d \leftarrow \infty$; $TG^i_k(v).pred \leftarrow null$;
    $TG^i_k(v).suc' \leftarrow TG^i_k(v).suc$;
    $TG^i_k(v).suc \leftarrow null$; $TG^i_k(v).nbr \leftarrow null$;
  }
  $TG^i_k(k).d \leftarrow 0$;
}
```

Figure 3.5: STAR Specification

```
BuildShortestPathTree($k, k'$)
{
  InitializeSingleSource($k$);
  $Q \leftarrow$ set of vertices in $TG^i_k$;
  $u \leftarrow$ ExtractMin($Q$); $newST \leftarrow \emptyset$;
  while ( $u \neq null$ and $TG^i_k(u).d < \infty$ )
  {
    if ( $TG^i_k(u).pred \neq null$ and $TG^i_k(u).pred \notin newST$ )
    {
      $(r, s) \leftarrow TG^i_k(u).pred$; $newST \leftarrow newST \cup (r, s)$;
      if ( LORA and $k = i$ )
      {
        if ( $k' \neq i$ and $TG^i_{k'}(u).suc = i$ and $TG_i(u).suc' = k'$ )
          $M_i \leftarrow$ TRUE;     // LORA-3 rule
        if ( $TG_i(u).suc \neq TG_i(u).suc'$ and $TG_i(u).suc > i$ )
          $M_i \leftarrow$ TRUE;      // LORA-3 rule
        if ( $\nexists\ (x, y) \in ST'_i \ \mid\ y = u$ )
          $M_i \leftarrow$ TRUE;      // LORA-1 rule
        if ( $\mid\ TG_i(u).d - TG_i(u).d''\ \mid\ > \Delta$ )
        {
          $M_i \leftarrow$ TRUE;      // LORA-2 rule
          $TG_i(u).d'' \leftarrow TG_i(u).d$;
        }
        if ( $k' \neq i$ and $TG^i_{k'}(u).pred =$ null )
          $M_i \leftarrow$ TRUE;      // LORA-2 rule
        $w \leftarrow TG_i(u).suc$;
        if ( $w \neq i$ )
          $path\_w\_u\_cost \leftarrow TG_i(u).d - TG_i(i, w).l$;
        else $path\_w\_u\_cost \leftarrow 0$;
        if ( $path\_w\_u\_cost > TG_i(u).d'$ )
        {
          if ( $r = w$ or $TG_i(r).nbr = i$ )
            $TG_i(s).nbr \leftarrow i$;
          if ( $TG_i(s).nbr \neq i$ )
            $M_i \leftarrow$ TRUE;      // LORA-3 rule
        }
        $TG_i(u).d' \leftarrow path\_w\_u\_cost$;
        $TG_i(u).suc' \leftarrow TG_i(u).suc$;
      }
    }
    for each ( vertex $v \in$ adjacency list of $TG^i_k(u)$
        $\mid TG^i_k(u, v).l \neq \infty$ and NOT $TG_i(u, v).del$ )
    {
      if ( $k = i$ )
      {
        if ( $u = i$ ) $suc \leftarrow i$;
        else if ( $TG_i(u).suc = i$ )
          $suc \leftarrow \{x \ \mid\ x \in N_i$ and $x = u\}$;
        else $suc \leftarrow TG_i(u).suc$;
      }
      else
      {
        if ( $u = k$ )
          if ( $v = i$ )     $suc \leftarrow i$;
          else               $suc \leftarrow k$;
        else $suc \leftarrow TG_i(u).suc$;
      }
      if ( $k \neq i$ or $u = i$ or $(u, v) \in ST^i_{suc}$ )
        RelaxEdge($k, u, v, Q, suc$);
    }
    if ( $Q \neq \emptyset$ )     $u \leftarrow$ ExtractMin($Q$);
    else              $u \leftarrow null$;
  }
  UpdateNeighborTree($k, newST$);
  if ( $k = i$ )
  {
    if ( LORA and $M_i$ )
    {
      ReportChanges($ST'_i, newST$);
      $ST'_i \leftarrow newST$;    $NS_i \leftarrow$ FALSE;
    }
    else if ( ORA )
      ReportChanges($ST_i, newST$);
    for each ( link $(u, v) \in TG_i$  $\mid$  $TG_i(u, v).del =$ TRUE )
      if ( ORA or ( LORA and ( $M_i$ or $(u, v) \notin ST'_i$ ) ) )
        $TG_i \leftarrow TG_i - \{(u, v)\}$;
    $M_i \leftarrow$ FALSE;
  }
  $ST^i_k \leftarrow newST$;    $newST \leftarrow \emptyset$;
}
```

```
RelaxEdge($k, u, v, Q, suc$)
{
  if ( $TG^i_k(v).d > TG^i_k(u).d + TG^i_k(u, v).l$ or
      ( $k = i$ and $TG^i_k(v).d = TG^i_k(u).d + TG^i_k(u, v).l$ and
        $(u, v) \in ST_i$ ) )
  {
    $TG^i_k(v).d \leftarrow TG^i_k(u).d + TG^i_k(u, v).l$;
    $TG^i_k(v).pred \leftarrow TG^i_k(u, v)$;
    $TG^i_k(v).suc \leftarrow suc$;
    if ( LORA and $k = i$ and $TG_i(v).suc' = null$ )
    {
      // $v$ was an unknown destination
      $TG_i(v).suc' \leftarrow suc$;
      $TG_i(v).d'' \leftarrow TG_i(v).d$;
      if ( $suc \neq i$ )
        $TG_i(v).d' \leftarrow TG_i(v).d - TG_i(i, suc).l$;
      else
        $TG_i(v).d' \leftarrow 0$;
    }
    Insert($Q, v$);
  }
}

ReportChanges($oldST, newST$)
description
  Generate LSUs for new links in the router's source tree
{
  for each ( link $(u, v) \in newST$ )
    if ( $(u, v) \notin oldST$ or $newST(u, v).t \neq oldST(u, v).t$
        or $NS_i$ )
      $MSG_i \leftarrow MSG_i \cup \{(\ u, v, TG_i(u, v).l, TG_i(u, v).t)\}$;
}

UpdateNeighborTree($k, newST$)
description
  Delete links from $TG^i_k$ and report failed links
{
  for each ( link $(u, v) \in ST^i_k$ )
  {
    if ( $(u, v) \notin newST$ )
    {
      // $k$ Has removed $(u, v)$ from its source tree
      if ( LORA and $TG^i_k(v).pred = null$ )
      {
        // LORA-2 rule: $k$ has no path to destination $v$
        $M_i \leftarrow$ TRUE;

        if ( $k = i$ )
          for each ( link $(r, s) \in TG_i \mid s = v$ )
            if ( $TG_i(r, s).l = \infty$ )
              $MSG_i \leftarrow MSG_i \cup \{(\ r, s, TG_i(r, s).l, TG_i(r, s).t)\}$;
      }
      if ( ORA and $k = i$ and ( $u = i$ or $TG_i(v).pred = null$ ) )
      {
        // $i$ has no path to destination $v$ or $i$ is the head node
        if ( $TG_i(v).pred = null$ )
          for each ( link $(r, s) \in TG_i \mid s = v$ )
            if ( $TG_i(r, s).l = \infty$ )
              $MSG_i \leftarrow MSG_i \cup \{(\ r, s, TG_i(r, s).l, TG_i(r, s).t)\}$;
        else if ( $TG_i(u, v).l = \infty$ )
          // $i$ Needs to report failed link
          $MSG_i \leftarrow MSG_i \cup \{(\ u, v, TG_i(u, v).l, TG_i(u, v).t)\}$;
      }
      if ( LORA and $k = i$ and $TG_i(v).pred = null$ )
      {
        $TG_i(v).d' \leftarrow \infty$;
        $TG_i(v).d'' \leftarrow \infty$;
        $TG_i(v).suc' \leftarrow null$;
      }
      if ( NOT ( $k = i$ and $u = i$ ) )
      {
        if ( $(u, v) \in TG^i_k$ )
          $TG^i_k \leftarrow TG^i_k - \{(u, v)\}$;
        if ( $k = i$ and $TG_i(u, v).l \neq \infty$ and $TG_i(u).pred \neq null$
            and $TG_i(v).pred = null$ )
          // $(u, v)$ is the root of an unreachable subtree
          $MSG_i \leftarrow MSG_i \cup \{(u, v, \infty, TG_i(u, v).t)\}$;
        if ( $TG_i(u, v).l \neq \infty$ and $\nexists\ x \in N_i \mid (u, v) \in TG^i_x$ )
          $TG_i(u, v).del \leftarrow$ TRUE;
      }
    }
  }
}
```

Figure 3.6: STAR Specification (cont.)

a current time stamp. The same approach is used for link failures (*NeighborDown*) and changes in link cost (*LinkCostChange*). When a router establishes connectivity to a new neighbor, the router sends its complete source tree to the neighbor (much like a distance vector protocol sends its complete routing table). The LSUs that must be broadcast by router i to all neighbors are inserted into the list MSG_i.

The procedure *Update* is executed when router i receives an update message from neighbor k or when the parameters of an outgoing link have changed. First, the topology graphs TG_i and TG_k^i are updated, then the source trees ST_k^i and ST_i are updated, which may cause the router to update its routing table and to send its own update message.

The state of a link in the topology graph TG_i is updated with the new parameters for the link if the link-state update in the received message is valid, i.e., if the LSU has a greater time stamp than the time stamp of the link stored in TG_i.

The parameters of a link in TG_k^i are always updated when processing an LSU sent by a neighbor k, even if the link-state information is outdated, because they report changes to the source tree of the neighbor. A node in a source tree ST_k^i can have only one link incident to it. Hence, when i receives an LSU for link (u, v) from k the current incident link (u', v) to v, $u \neq u'$, is deleted from TG_k^i.

The information of an LSU reporting the failure of a link is discarded if the link is not in the topology graph of the router.

A shortest-path algorithm based on Dijkstra's SPF (procedure *BuildShortestPathTree*) is run on the updated topology graph TG_k^i to construct a new source tree ST_k^i, and then run on the topology graph TG_i to construct a new source tree ST_i.

The incident link to a node v in router's i new source tree is different from the link in the current source tree ST_i only if the cost of the path to v has decreased or if the incident link in ST_i was deleted from the source trees of all neighbors.

A new source tree $newST$ for a neighbor k, including the router's new source tree, is then compared to the current source tree ST^i_k (procedure *UpdateNeighborTree*), and the links that are in ST^i_k but not in $newST$ are deleted from TG^i_k. After deleting a link (u, v) from TG^i_k the router sets $TG_i(u, v).del$ to TRUE if the link is not present in the topology graphs $TG^i_x, \forall x \in N_i$.

If a destination v becomes unreachable, i.e., there is no path to v in the new source tree $newST$, then LSUs will be broadcast to the neighbors for each link in the topology graph TG_i that have v as the tail node of the link and a link cost infinity.

For simplicity, this specification assumes that the link layer provides reliable broadcast of control packets and consequently update messages specify only incremental changes to the router's source tree instead of the complete source tree.

The new router's source tree $newST$ is compared to the last reported source tree ($ST_i{}'$ for LORA and ST_i for ORA) (procedure *ReportChanges*), and an update message that will be broadcast to the neighbors is constructed from the differences of the two trees. An LSU is generated if the link is in the new source tree but not in the current source tree, or if the parameters of the link have changed. For the case of a router running LORA, the source trees are only compared with each other if at least one of the three conditions (LORA-1, LORA-2, and LORA-3) described in Section 3.2.5 is met, i.e., M_i = TRUE.

If the new router's source tree was compared against the last reported source tree then the router removes from the topology graph all the links that are no longer used by any neighbor in their source trees.

Finally, the current shortest-path tree ST^i_k is discarded and the new one becomes the current source tree. The router's source tree is then used to compute the new routing table, using for example a depth-first search in the shortest-path tree.

3.3 STAR Correctness

For simplicity, we assume that all links are point-to-point and that shortest-path routing is implemented. Let t_0 be the time when the last of a finite number of link-cost changes occur, after which no more such changes occurs. The network $G = (V, E)$ in which STAR is executed has a finite number of nodes ($\mid V \mid$) and links ($\mid E \mid$), and every message exchanged between any two routers is received correctly within a finite time. According to STAR's operation, for each direction of a link in G, there is a router that detects any change in the cost of the link within a finite time.

For simplicity, the following theorems assume that link-state information does not age-out.

3.3.1 Correctness of STAR Under ORA

This section addresses STAR correctness for the case in which ORA is applied. The following theorems show that routers running STAR under ORA send correct routing tables within a finite time after link costs change, without the need to replicate topology information at every router (like the Open Shortest Path First (OSPF) protocol does) or use explicit delete updates to delete obsolete information (like the Link Vector Algorithm (LVA) [19] and the Adaptive Link-State Protocol (ALP) [20] do).

Theorem 6 *The dissemination of LSUs by routers running STAR under ORA stops a finite time after t_0.*

Proof: A router that detects a change in the cost of any outgoing link must update its topology graph, update its source tree as needed, and send an LSU if the link is added to or is updated in the source tree. Let l be the link that last experiences a cost change up to t_0, and let t_l be the time when the head of the link l originates the last LSU originated as a result of the link-cost changes occurring up to t_0. Any router that receives the LSU for link l originated at t_l must process the LSU within a finite time, and decides whether or not to forward the LSU based on its updates to its source tree. A router can accept and propagate an LSU only once because each LSU has a time stamp; accordingly, given that G is finite, there can only be a finite chain of routers that propagate the LSU for link l originated at t_l, and the same applies to any LSU originated from the finite number of link-cost changes that occur up to t_0. Therefore, STAR stops the dissemination of LSUs a finite time after t_0. □

Because of Theorem 6, there must be a time $t_s > t_0$ when no more LSUs are being sent.

Theorem 7 *When ORA is applied, in a connected network, and in the absence of link failures, all routers have the most up-to-date link-states they need to compute shortest paths to all destinations within a finite time after t_s.*

Proof: The proof is by induction on the number of hops of a shortest path to a destination (the origin of an LSU), and is similar to the proof for SPTA [3].

Consider the shortest path from router s_0 to a destination j at time t_s, and let h be the number of hops along such a path. For $h = 1$, the path from s_0 to j consists of one of the router's outgoing links. By assumption, an underlying protocol informs the router with the correct parameter values of adjacent links within a finite time; therefore, the Theorem is true for $h = 1$, i.e., s_0 must have link (s_0, j) in its source tree and must have sent its neighbors an LSU for that link. Assume that the Theorem is true for $h = n$ hops, i.e., that any router with a path of n or fewer hops to j has the correct link-state information about all the links in the path and has sent LSUs to its neighbors with the most up-to-date state of each such link, and consider the case in which the path from s_0 to j at time t_s is $n+1$ hops. Let s_1 be the next hop along the shortest path selected by s_0 to j at t_s. The sub-path from s_1 to j has n hops and, by the inductive assumption, such a path must be in the source tree of s_1 at time t_s, which implies that all the links in the path from s_1 to j are in its source tree, and that s_0 received and processed an LSU for each link in the path from s_1 to j with the most recent link-state information. Because it is also true that s_0 has the most recent link-state information about link (s_0, s_1), it follows that s_0 has the most recent information about all the links in its chosen path to j. The Theorem is therefore true, because the same argument applies to any chosen destination and router. □

Theorem 8 *When ORA is applied, all the routers of a connected network have the most up-to-date link-state information needed to compute shortest paths to all destinations.*

Proof: The result is immediate from Theorem 7 in the absence of link failures. Consider the case in which the only link that fails in the network by time t_0 is link (s, d). Call the time when link (s, d) fails $t_f \leq t_0$. According to STAR's operation, router s sends an LSU reporting an infinite cost for (s, d) within a finite time after t_f; furthermore, every router receiving the LSU reporting the infinite cost of (s, d) must forward the LSU if the link exists in its topology graph, i.e., the LSU gets flooded to all routers in the network that had heard about the link, and this occurs within a finite time after t_0; moreover, links with infinite cost are not erased from the topology graph. It then follows that no router in the network can use link (s, d) for any shortest path within a finite time after t_0. Accordingly, within a finite time after t_0 all routers must only use links of finite cost in their source trees; together with Theorem 7, this implies that the Theorem is true. □

Theorem 9 *When ORA is applied, if destination j becomes unreachable from a network component C at t_0; the topology graph of all routers in C includes no finite-length path to j.*

Proof: STAR's operation is such that, when a link fails, its head node reports an LSU with an infinite cost to its neighbors, and the state of a failed link is flooded through a connected component of the network to all those routers that knew about the link. Because a node failure equals the failure of all its adjacent links, and because every neighbor of a failed node must detect the failure of its link to the node within a finite time, it is true that no router in C can compute a finite-length path to j from its topology graph after a finite time after t_0. □

Note that, if a connected component remains disconnected from a destination j all link-state information corresponding to links for which j is the head node is erased from the topology graph of the routers in the network component.

3.3.2 Correctness of STAR Under LORA

This section addresses the correctness of STAR for the case in which LORA is applied. Correctness under LORA simply means that routers stop sending updates and the source trees at the routers of a network do not imply routing table loops, include no paths to unreachable destinations, and span all reachable destinations within a finite time after the moment when no more topology changes or link cost changes occur. Assume that the successor entries of the nodal routing tables in G define another graph for each destination j, denoted $S_j(G)$, whose nodes are the same nodes of G and in which a directed edge exists from node i to node k if and only if node k is i's successor to j. Also assume that P_{xj} denotes the path from node x to node j in $S_j(G)$. In steady-state, loop freedom is guaranteed in G if $S_j(G)$ is a tree within a finite time. Node i is said to be *upstream* of node k in $S_j(G)$ if the directed chain P_{ij} from node i to node j includes node k. Similarly, node k is *downstream* of node i.

For simplicity, we assume that $\Delta = \infty$. As we described in Section 3.3.1, we assume that there is a time t_0 after which no topology changes occur. The following theorems show that routers running STAR under LORA set correct routing tables within a finite time after link cost change without incurring permanent loops.

Theorem 10 *The dissemination of LSUs by routers running STAR under LORA stops a finite time after t_0.*

Proof: A router that detects a change in the cost of any outgoing link must update its topology graph, update its source tree as needed, and sends an LSU only if a new destination is found, a destination becomes unreachable, or a permanent loop can be formed, which is a subset of the cases under ORA. Accordingly, the result is immediate from Theorem 6. □

Because of Theorem 10, there must be a time $t_s > t_0$ when no more LSUs are being sent.

Theorem 11 *The path to a destination j is loop-free within a finite time after t_0 if a node sends an update message reporting changes to its source tree whenever the distance implied in the source tree of the new chosen successor s for a given destination j is longer than the distance implied in the source tree from the previous successor for the same destination.*

Proof: Assume that a router i running STAR only reports changes to its source tree when i finds a new destination, a destination becomes unreachable, or the cost of the path to j reported by the new successor is larger than the cost of the path to j reported by the previous successor.

Let $l = (i, x)$ be the link that last experiences a cost change at t_0, and assume that the cost of l has decreased. If the distance implied in the reported source tree of the new successor s to x is shorter or equal to the distance implied in the source tree of the previous successor, then it is true that s is downstream of node i in $S_x(G)$ and therefore no permanent loop can be formed. This is also true for any destination j that has l in $S_j(G)$.

On the other hand, if the cost of the link l had increased and the distance reported by the new successor s to x is longer than the reported distance from the previous successor to x, then i generates an update message reporting the changes to its source tree. In turn, upstream nodes that have i in their paths to x must either experience a distance increase and send an update, or find an alternate loop-free path to x of the same length. From Theorem 10, the last update message is processed within a finite time after t_0. □

Theorem 12 *No permanent loop can be formed if no update message is generated by a node i regarding destination j or any destination whose path from i involves j when the new successor $s \neq i$ to j is a neighbor of j and $i > s$.*

Proof: The proof is by contradiction. Consider that at time $t_i < t_0$ the cost of the link $(i, j) \in ST_i$ changes and node i does not generate an update message after choosing $s \in N_j$ as the new successor to j. Also, assume that a permanent loop to j between i and s is formed at time t_0 when the cost of the link (s, j) changes and s chooses $i \in N_j$ as the successor to j without reporting changes to its source tree. According to STAR's operation, if at time t_i node i did not generate an update message, then $i > s$. This implies that at time t_0 node s should have generated an update message, because $s < i$, which contradicts the Theorem. According to STAR's operation, router i cannot add a link (u, v) to its new source tree choosing neighbor s as the successor to v if (u, v) is not in the source tree reported by s. Given that the update message transmitted by s reports the addition of the link (i, j) to its source tree and the implicit deletion of the link (s, j), node i cannot choose s as its successor towards j and consequently no permanent loop can be formed. □

Theorem 13 *thm8 When LORA is applied, all the routers in a connected network C have a loop-free path to every destination in C within a finite time after t_0.*

Proof: According to STAR's operation, an LSU reporting a new destination is flooded by all the routers in C. Moreover, a router reports the changes to its new source tree to a neighbor if the router has a path to a destination in C that became unreachable to the neighbor. Therefore, a finite-length path to any destination in a connected network C is known by all the routers in C. □

Theorem 14 *When LORA is applied, if destination j becomes unreachable from a network component C at t_0; the topology graph of all routers in C includes no finite-length path to j.*

Proof: STAR's operation is such that, when a node i in C finds that j has become unreachable, i reports an LSU to its neighbors for each link in the topology graph that has an infinite cost and j as the tail node. The state of the failed links is flooded through a connected component of the network to all those routers that knew about j and find j unreachable. Because a node failure equals the failure of all its adjacent links, it is true that no router in C can compute a finite-length path to j from its topology graph after a finite time after t_0. □

3.4 Performance Evaluation

3.4.1 Simulation Experiments

STAR has the same communication, storage, and time complexity than ALP [20] and efficient table-driven distance-vector routing protocols proposed to date (e.g., WRP [36]). However, worst-case performance is not truly indicative of STAR's performance; accordingly, we ran a number of simulation experiments to compare STAR's average performance against the performance of table-driven and on-demand routing protocols.

The protocol stack implementation in our simulator runs the very same code used in a real embedded wireless router and IP (Internet Protocol) is used as the network protocol.

The link layer implements a medium access control (MAC) protocol similar to the IEEE 802.11 standard and the physical layer is based on a direct sequence spread spectrum radio with a link bandwidth of 1 Mbit/sec. An underlying protocol is configured to report loss of connectivity to a neighbor if the quality of the link with the neighbor decreases to unacceptable levels in a period of about 10 seconds.

The simulation experiments use 20 nodes forming an ad hoc network, moving over a flat space (5000m x 7000m), and initially randomly distributed at a density of one node per square kilometer. Nodes move in the simulation according to the "random waypoint" model [11]. Each node begins the simulation by remaining stationary for *pause time* seconds. It then selects a random destination and moves to that destination at a speed of 20 meters per second for a period of time uniformly distributed between 5 and 11 seconds. Upon reaching the destination, the node pauses again for *pause time* seconds, selects another destination, and proceeds there as previously described, repeating this behavior for the duration of the simulation.

The simulation study was conducted in the C++ Protocol Toolkit (CPT) simulator environment [4]. STAR based on ORA is first compared against two other table-driven routing protocols, and STAR based on LORA is then compared with DSR, which has been shown to be a very bandwidth efficient on-demand routing protocol.

3.4.2 Comparison with Table-Driven Protocols

We chose to compare STAR against the traditional link-state approach and ALP [20]. The traditional link-state approach (denoted by TOB for topology broadcast) corresponds to the flooding of link states in a network, or within clusters coupled with flooding of inter-cluster connectivity among clusters; a link-state update is flooded throughout the network only when the cost of the link changes. ALP is a routing protocol based on partial link-state information that we have previously shown to outperform prior table-driven distance-vector and link-state protocols [20]. For these simulations STAR uses ORA, because both ALP and TOB attempt to provide shortest paths. In the simulation experiment, the three protocols rely on the reliable delivery of broadcast packets by the link layer. As such, the results presented present the best possible behavior for any link-state protocol based on flooding, and the best possible behavior for ALP and STAR.

We ran our simulations with movement patterns generated for 5 different pause times: 0, 30, 45, 60, and 90 seconds. A pause time of 0 seconds corresponds to continuous motion. The simulation time in all the simulation scenarios is of 900 seconds.

As the pause time increases, we expect the number of update packets sent to decrease because the number of link connectivity changes decreases. Because STAR and ALP generate LSUs for only those links along paths used to reach destinations, we expect STAR and ALP to outperform any topology broadcast protocol.

A link in the topology graph of a router running ALP can be in one of three states: state 0 if the link is in the topology graph but is not in the preferred paths of the router, state 1 if the link is in the preferred paths of the router, and state 2 if the link was deleted from the preferred paths of the router and the cost of the link has not increased at the time of its deletion from the preferred paths. A router running ALP does not report to its neighbors the deletion of a link from its preferred paths if the cost of the link has not increased, i.e., when the state of the link in the topology graph transitions from 1 to 2. Consequently, all the routers that have a link in state 2 in their topology graphs have to forward to their neighbors an LSU that announces the failure of the link. Unlike ALP, routers running STAR only have in their topology graphs link-state information for those links that are in the preferred paths

Pause Time	Connectivity Changes	Update Packets Generated		
		STAR	ALP	TOB
0	1090	2542	–	–
30	154	411	1765	5577
45	102	262	1304	3908
60	90	239	1144	2502
90	50	138	623	1811

Table 3.1: Average performance of STAR, ALP, and TOB.

of their neighbors, i.e., the failure of a link will only make a router send an update message reporting the failure if the link is in the preferred paths of the router and a destination becomes unreachable. The head node of a failed link must report the failure of the link both in STAR and ALP.

Table 3.1 summarizes the behavior of the three protocols according to the *pause time* of the nodes. The table shows the number of link connectivity changes and the total number of update packets generated by the routing protocols; ALP generates on average more than 4 times more update packets than STAR, and topology broadcast generates more than 10 times more packets than STAR. The performance of ALP and TOB for *pause time* 0 could not be assessed because the amount of update packets generated by the routers lead to congestion at the link layer.

Because STAR can be used in combination with any clustering scheme proposed in the past for packet-radio networks, it is clear from this study that STAR should be used instead of ALP and topology broadcast for the provision of QoS routing in packet radio networks, given that any overhead traffic associated with clustering would be equivalent for STAR, ALP, and topology broadcast.

3.4.3 Comparison with DSR

As we have stated, our simulation experiments use the same methodology used recently to evaluate DSR and other on-demand routing protocols [11]. To run DSR in our simulation environment, we ported the ns2 code available from [45] into the CPT simulator. There are only two differences in our DSR implementation with respect to that used in [11]: (1) there is no access to the MAC layer in the embedded wireless routers and simulated protocol stack we used, which implies that packets already scheduled for transmission over a link cannot be rescheduled in either protocol, and (2) routers cannot operate their network interfaces in *promiscuous mode* because the MAC protocol operates over multiple channels and a router does not know on which channels its neighbors are transmitting, unless the packets are meant for the router. Both STAR and DSR can buffer 20 packets that are awaiting discovery of a route through the network.

The overall goal of the simulation experiments was to measure the ability of the routing protocols to react to changes in the network topology while delivering data packets to their destinations. The data traffic load was kept small to ensure that, if links were congested, it was due to control traffic. We applied to the simulated network three different communication patterns corresponding to 8, 14, and 20 data flows. The total workload in the three scenarios was the same and consisted of 32 data packets/sec, in the scenario with 8 flows each continuous bit rate (CBR) source generated 4 packets/sec, in the scenario with 20 sources each CBR source generated 1.6 packets/sec, and in the scenario with 14 flows there were 7 flows from distinct CBR sources to the same destination D generating an aggregate of 4 packets/sec and 7 flows having D as the CBR source and the other 7 sources of data as destinations. In all the scenarios the number of unique destinations was 8 and the packet size was 64 bytes. The data flows were started at times uniformly distributed between 20 and 120 seconds (we chose to start the flows after 20 seconds of simulated time to give some time to the Link Layer for determining the set of nodes that are neighbors of the routers).

Since the performance of STAR depends on the type of service provided by the link-layer, we run two sets of simulation experiments. In the first experiment the MAC protocol ensures the reliable broadcast of control packets to all the neighbors of a router and, thus, allowing routers running STAR to send incremental updates. In the second experiment routers running STAR send their complete source tree in every update message and exchange update messages according to LORA rules LORA-1 to LORA-7.

3.4.4 Comparison with DSR Using Reliable Broadcasts

The protocol evaluations are based on the simulation of 20 wireless nodes in continuous motion (*pause time* 0) for 900 seconds of simulated time. The total number of changes in link connectivity is 1460.

Table 3.2 summarizes the behavior of STAR and DSR. The table show the total number of update packets transmitted by the nodes and the total number of data packets delivered to the applications for the three simulated workloads. The total number of update packets transmitted by routers running STAR varies with the number of changes in link connectivity while DSR generates control packets based on both variation of changes in connectivity and the type of workload inserted in the network. Routers running STAR generated fewer update packets than DSR in all simulated scenarios: routers running DSR sent from 0.35 to 4.40 times more update packets. Both STAR and DSR were able to deliver about the same number of data packets to the applications in the simulated scenarios with 8 and 14 flows. When we increased the number of sources of data from 8 to 20 nodes, while inserting the same number of data packets in the network (32 packets/sec), we observed that STAR was able to deliver

Number	Update Pkts Sent		Data Pkts Delivered		Data Pkts
of Flows	STAR	DSR	STAR	DSR	Generated
8	585	791	14898	14740	24100
14	560	1466	15206	15367	25917
20	575	3122	13922	6830	23718

Table 3.2: Average performance of STAR and DSR

Number	Protocol	Number of Hops					
of Flows		1	2	3	4	5	6
8	STAR	92.0	7.4	0.2		0.4	
	DSR	64.9	31.2	2.6	1.3		
14	STAR	82.0	16.0	1.7		0.3	
	DSR	64.1	26.9	4.0	4.5	0.5	
20	STAR	92.6	5.1	2.0	0.3		
	DSR	61.9	32.4	5.1	0.3		0.3

Table 3.3: Distribution of DATA packets delivered according to the number of hops traversed from the source to the destination

twice the amount of data packets delivered by DSR during 900 seconds of simulated time.

The MAC layer discards all packets scheduled for transmission to a neighbor when the link to the neighbor fails, which contributes to the high loss of data packets seen by nodes. In DSR, each packet header carries the complete ordered list of routers through which the packet must pass and may be updated by nodes along the path towards the destination. The low throughput achieved by DSR for the case of 20 sources of data is due to the poor choice of source routes the routers make, leading to a significant increase in the number of ROUTE ERROR packets generated. Data packets are also discarded due to lack of routes to the destinations, because the network may become temporarily partitioned or because the routing tables have not converged in the highly dynamic topology we simulate.

Figures 3.7(a) through 3.7(c) show the cumulative distribution of packet delay experienced by data packets during 900 seconds of simulated time, for a workload of 8, 14, and 20 flows, respectively. The higher delay introduced by DSR when relaying data packets is not directly related with the number of hops traversed by the packets (as shown in Table 3.3), but with the use of stale source routes when the number of flows increase from 8 to 20.

In all the simulation scenarios the number of destinations was set to just 40% of the number of nodes in the network in order to be fair with DSR. For the cases in which all the nodes in the network receive data, STAR would introduce no extra overhead while DSR could be severely penalized. It is also important to note the low ratio of update messages generated by STAR compared to the number of changes in link connectivity (Table 3.2).

We note that in cases where routers fail or the network becomes partitioned for extended time periods, the bandwidth consumed by STAR is much the same as in scenarios in which no router fails, because all that must happen is for updates about the failed links to unreachable destinations to propagate across the network. In contrast, DSR and several other on-demand routing protocols would continue to send flood-search messages trying to reach the failed destination, which would cause a worst-case bandwidth utilization for DSR. To illustrate the impact the failure of a single destination has in DSR we have run the simulation scenario with 8 flows present in the network for 1800 seconds making one of the destinations fail after 900 seconds of simulated time, routers running STAR sent 942 (946 without the node failure) update packets while routers running DSR sent 3043 (1963 without the node failure) update packets. The existence of a single flow of data to a destination that was unreachable for 900 seconds made DSR to generate 55% more update packets while STAR generated about the same number of updates.

3.4.5 Comparison with DSR Using Rules LORA-1 to LORA-7

When routers exchange update messages according to rules LORA-1 to LORA-7 the performance of STAR depends on the type of workload inserted in the network. For this reason, the protocol evaluations are based on simulations with movement patterns generated for five

Pause Time	Num. Flows	Update Pkts Sent		Data Pkts Delivered		Data Pkts Generated
		STAR	DSR	STAR	DSR	
0	8	908	791	15110	14740	24100
	14	930	1460	15845	10975	25917
	20	916	3122	13689	6830	23718
15	8	615	460	19544	20831	24396
	14	636	702	23027	23210	25989
	20	686	1535	17254	10129	23649
30	8	559	350	20180	20492	24160
	14	551	464	23086	23228	25892
	20	580	763	19929	18341	23716
45	8	517	280	21685	22683	24100
	14	526	2352	23776	20481	25917
	20	507	1880	20749	19898	23731
60	8	522	482	22536	19102	24100
	14	507	1357	24473	23436	25917
	20	493	744	22218	21899	23775

Table 3.4: Performance of STAR and DSR

different pause times: 0, 15, 30, 45, and 60 seconds. The experiments are based on a 20-node network with a simulated time of 900 seconds.

Table 3.4 summarizes the behavior of STAR and DSR. The table shows the total number of update packets transmitted by the nodes and the total number of data packets delivered to the applications for the three simulated workloads. Table 3.5 shows the number of hops traversed by data packets when nodes are in continuous motion. Routers running STAR generate fewer update packets than DSR in most of the simulated scenarios, the difference increases significantly when the number of flows in the network is 20 (routers running DSR sent up to two times more control packets than STAR when nodes were in continuous motion). Both STAR and DSR were able to deliver about the same number of data packets to the applications in the simulated scenarios with 8 and 14 flows. When we increase the number of sources of data from 8 to 20 nodes, while inserting the same number of data packets in the network (32 packets/sec), we observe that STAR is able to deliver as much as twice the amount of data packets delivered by DSR when nodes are in continuous motion. It is also important to

Number of Flows	Protocol	Number of Hops					
		1	2	3	4	5	6
8	STAR	94.0	4.1	1.9			
	DSR	64.9	31.2	2.6	1.3		
14	STAR	76.0	16.4	4.2	3.0	0.4	
	DSR	64.1	26.9	4.0	4.5	0.5	
20	STAR	90.8	6.4	1.6	1.1	0.1	
	DSR	61.9	32.4	5.1	0.3		0.3

Table 3.5: Number of hops traversed by data packets (pause time 0)

Pause Time	Connectivity Changes
0	1461
15	605
30	424
45	350
60	322

Table 3.6: Changes in link connectivity

note the low ratio of update messages generated by STAR compared to the number of changes in link connectivity (Table 3.6).

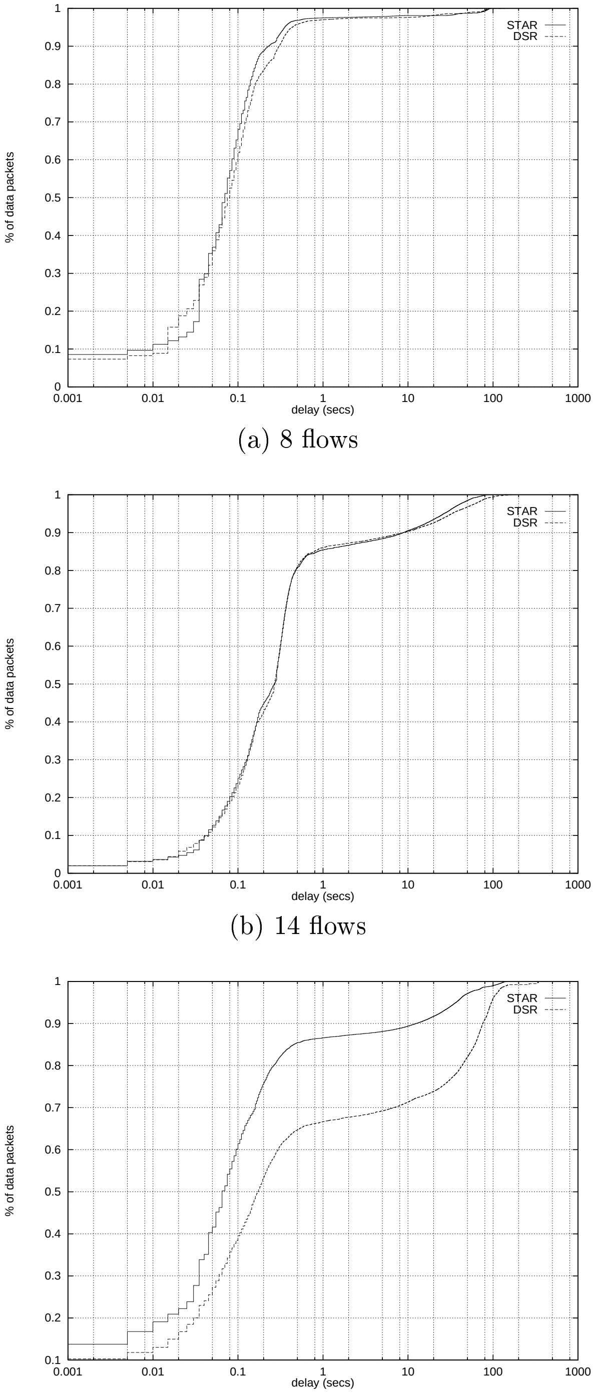
Figures 3.8(a) through 3.8(c) show the cumulative distribution of packet delay experienced by data packets when nodes are in continuous motion for a workload of 8, 14, and 20 flows respectively. We note that the distribution of the latency is about the same for both STAR and DSR.

To illustrate the impact the failure of a single destination has in DSR we have run the simulation scenario with 8 flows present in the network for 1800 seconds making one of the destinations fail after 900 seconds of simulated time, routers running STAR sent 1823 (1583 without the node failure) update packets while routers running DSR sent 3043 (1963 without the node failure) update packets. The existence of a single flow of data to a destination that was unreachable for 900 seconds made DSR generate 55% more update packets while STAR experienced an increase of 15%.

3.5 Conclusions

We have presented STAR, a link-state protocol that incurs smaller communication overhead than the ideal topology broadcast protocol and ALP (which in turn has been shown to incur less communication overhead than protocols based on the Distributed Bellman Ford algorithm and LVA), and also incurs less overhead than one of the most bandwidth-efficient on-demand routing protocols proposed to date. STAR accomplishes its bandwidth efficiency by: (a) disseminating only that link-state data needed for routers to reach destinations; (b) exploiting that information to ascertain when update messages must be transmitted to detect new destinations, unreachable destinations, and loops; and (c) allowing paths to deviate from the ideal optimum while not creating permanent loops. The bandwidth efficiency achieved in STAR is critical for ad hoc networks with energy constraints, because it permits routers to preserve battery life for the transmission of user data while avoiding long-term routing loops or the transmission of data packets to unreachable destinations.

Our simulation experiments show that STAR is an order of magnitude more efficient than the traditional link-state approach, and more than four times more efficient than ALP (which has been shown to outperform prior topology-driven protocols), in terms of the number of update packets sent. The results of our experiments also show that STAR is 1.3 to 6 times more bandwidth efficient than DSR, which in turn has been shown to be one of the most bandwidth-efficient on-demand routing protocols. Because STAR can be used with any clustering mechanism proposed to date, these results clearly indicate that STAR is a very attractive approach for routing in packet-radio networks. Perhaps more importantly, the approach we have introduced in STAR for least-overhead routing opens up many research avenues, such as developing similar protocols based on distance vectors and determining how route aggregation and geographical routing works under LORA.

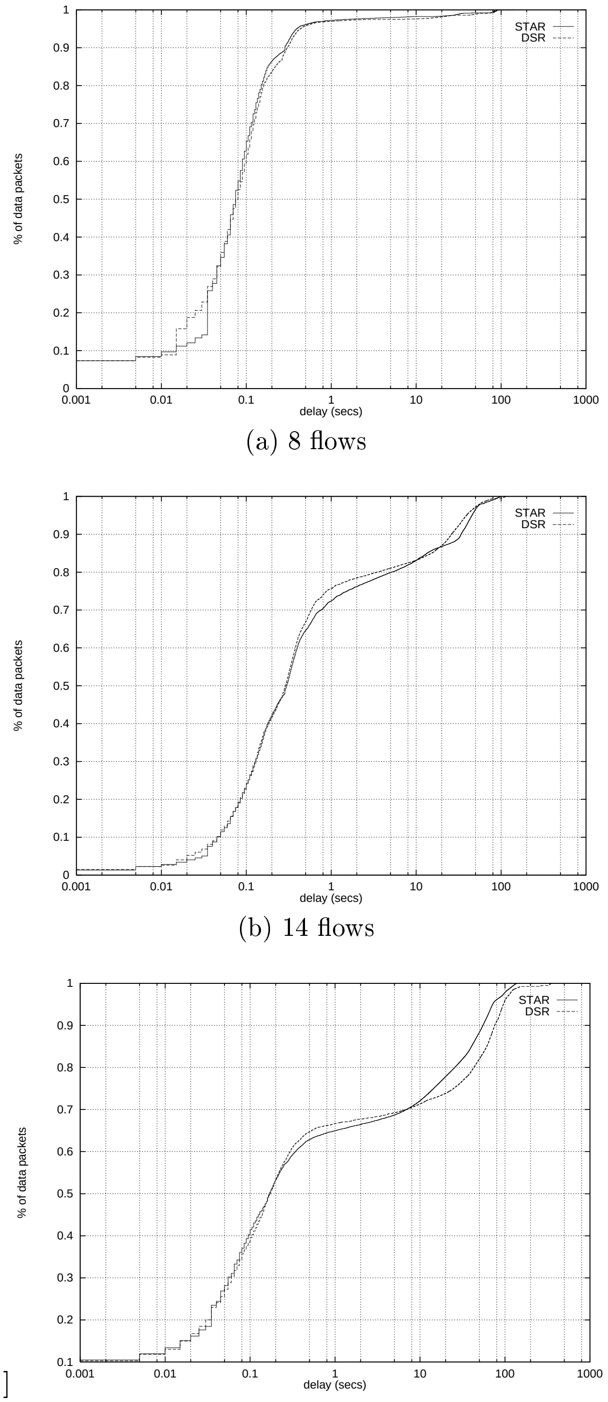


(a) 8 flows

(b) 14 flows

(c) 20 flows

Figure 3.7: Cumulative distribution of packet delay experienced by data packets



(a) 8 flows

(b) 14 flows

(c) 20 flows

Figure 3.8: Cumulative distribution of packet delay experienced by data packets

Chapter 4

Neighborhood Aware Source Routing

On-demand routing protocols have been shown to be very effective for ad hoc networks. The success of a caching algorithm for an on-demand routing protocol depends of the strategies used for the deletion of links from the routing cache [27]. DSR has been shown to incur less routing overhead when utilizing a cache data structure based on a graph representation of individual links (link cache), rather than based on complete paths. DSR removes failed links from the link cache when a ROUTE ERROR packet reports the failure of the link and other links are removed by aging. The lifetime of a link is estimated based on a node's perceived stability of both endpoint nodes of the link. In some scenarios, the use of link caches has been shown to produce overhead traffic as little as 50% of that incurred with path caches [27].

The results reported on DSR indicate that a routing protocol based on link-state information can make better routing decisions than one based on path information, because the freshness of routing information being processed can be determined by the timestamp or sequence number assigned by the head node of the links. On the other hand, in an ad hoc network, it is very easy for a given node to learn about the neighbors of its neighbors. Based on these observations, we introduce a new approach to link-state routing in ad hoc networks based on on-demand source routing and knowledge of links that exist in the two-hop neighborhood of nodes. We call this approach the *neighborhood aware source routing* (NSR) protocol. In NSR, a node maintains a partial topology of the network consisting of the links to its immediate neighbors (1-hop neighbors), the links to its 2-hop neighbors, and the links in the requested paths to destinations that are more than two hops away. Links are removed from this partial topology graph by aging only, and the lifetime of a link is determined by the node from which the link starts (head node of the link), reflecting with a good degree of certainty the degree of mobility of the node.

Section 4.1 describes NSR in detail. Section 4.2 demonstrates that routers executing NSR discover source routes to any destination in a connected network within a finite time, that the source of data packets is notified within a finite time when the destination becomes unreachable, that data packets cannot form any routing loop, and that NSR achieves correct reset of sequence numbers. Section 4.3 presents the performance comparison of NSR and DSR using a 50-node network and eight simulation experiments with different numbers of sources and destinations. The simulation results indicate that NSR incurs far less communication overhead while delivering packets with the same or better delivery rates and average delays as DSR using path caches.

4.1 NSR Description

4.1.1 Overview

To describe NSR, the topology of a network is modeled as a directed graph, where each node in the graph has a unique identifier and represents a router, and where the links connecting the nodes are described by some parameters. A link from node u to node v is denoted (u, v), node u is referred as the head node of the link and node v is referred as the tail node of the link. In this study it is assumed that the links have unit cost.

Routers are assumed to operate correctly and information is assumed to be stored without errors. All events are processed one at a time within a finite time and in the order in which they are detected. Broadcast packets are transmitted unreliably and it is assumed that the link-level protocol can inform NSR when a packet cannot be sent over a particular link.

The *source route* contained in a data packet specifies the sequence of nodes to be traversed by the packet. A source route can be changed by the routers along the path to the destination and its maximum length is bounded.

NSR does not attempt to maintain routes from every node to every other node in the network. Routes are discovered on an on-demand basis and are maintained only as long as they are necessary.

Routers running NSR exchange link-state information and source routes for all known destinations are computed by running Dijkstra's shortest-path first on the partial topology information (*topology graph*) maintained by the router. If NSR has a route to the destination of a locally generated data packet, a source route is added to the header of the packet and it is forwarded to the next hop. Otherwise, NSR broadcast a *route request* (RREQ) to its neighbors asking for the link-state information needed to build a source route to the destination. If the neighbors do not have a path, a RREQ is broadcast to the entire network and only the destination of the data packet is allowed to send to the source of the RREQ a *route reply* (RREP) containing the complete path to the destination. A router forwarding a data packet needs to send a *route error* (RERR) packet to the source of the data only when it is not able to find an alternate path to a broken source route or when the failure of a link in the source route needs to be reported to the source of the data packet in order to erase outdated link-state information.

NSR has the property of being loop-free at all times because any change made to the path to be traversed by a data packet does not include nodes from the traversed path.

A link to a new neighbor is brought up when NSR receives any packet from the neighbor. The link to a neighbor is taken down either when the link-level protocol is unable to deliver unicast packets to the neighbor or when a timeout has elapsed from the last time the router received any traffic from the neighbor.

A node running NSR periodically broadcasts to its neighboring nodes a *hello* (HELLO) packet. HELLO packets have a dual purpose: they are used to notify the presence of the node to its neighbors and as a way of obtaining reasonable up-to-date information about the set of links two hops away from the node. By having such link-state information refreshed periodically the nodes forwarding a data packet may not need to notify the source of the packet when a repair is done to a broken source route. As an example, the links shown as solid lines in Figure 4.1 represent the source route followed by a data packet sent by node a to node e, a link shown as a dashed line is known by the head node and its neighbors. Suppose links (b, c) and (c, d) fail and node a sends a data packet. NSR is able to repair the source route twice, without needing to report the link failures to node a, by making node b to add node f to the source route and by making node c to replace node d by node g.

NSR uses *sequence numbers* to validate link-state information. All the outgoing links

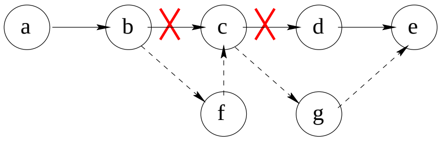


Figure 4.1: Failure of links (b, c) and (c, d) do not cause the generation of RERR packets when repairing the source route represented by the links in solid lines

of a node are identified by the same monotonically increasing sequence number. A node increments its sequence number when it needs to send a control packet and a link was brought up or taken down since the last time a control packet was transmitted.

Link-state information is only deleted from the topology graph due to aging. The lifetime of a link is determined by the head node of the link. All the outgoing links of a node are considered to have the same lifetime, which is computed according to a function that estimates the average time a link to a neighbor is up.

4.1.2 Routing Information Maintained in NSR

A node i running NSR maintains the node's epoch E_i, the node's sequence number SN_i, the average lifetime of the node's links to neighbors L_i, the broadcast ID, the neighbor table, the topology graph, the shortest-path tree, the data queue, the RREQ history table, and the RERR history table.

The node's epoch is incremented when the node boots up before NSR starts its operation. The node's epoch is the only data that needs to be kept in non-volatile storage because it is used to maintain the integrity of routing information across node's resets (a separate section describes its role in NSR's operation).

The node's sequence number is used in the validation of link-state information. The sequence number is incremented when the node needs to advertise the state of one of its outgoing links and one of the node's outgoing links changed state since the last time a link-state was advertised.

The lifetime of the node's links to neighbors is updated periodically by applying a decay factor to the current lifetime and averaging the time the links to neighbors are up.

The broadcast ID is used with the node's address to uniquely identify a RREQ packet. The broadcast ID has its value incremented when a new RREQ packet is created.

An entry in the neighbor table contains the address of a neighbor, the time the link to the neighbor was brought up, the *neighbor ID*, and a *delete flag*. When the link to a neighbor is taken down the *delete flag* is set to 1 to mark the entry as deleted. The neighbor table has 255 entries, and the *neighbor ID* consists of the number of the entry in the table. An entry marked as deleted is reused when the link for the neighbor listed in the entry is brought up. This guarantees that the *neighbor ID* is preserved for some time across link failures, which is useful when building source routes based on this IDs, as described later.

The topology graph is updated with the state of the links reported in both control packets and data packets. The parameters of each link (u, v) in the topology graph consists of the tuple (sn, $cost$, $lifetime$, $ageTime$, $nbrID$), where sn is the link's sequence number, $cost$ is the cost of the link, $lifetime$ is the link's lifetime as reported by u, $ageTime$ is the age-out time of the link, and $nbrID$ is the *neighbor ID* assigned by node u to its neighbor v.

The shortest-path tree is obtained by running Dijkstra's shortest-path on the topology graph when a control packet is received, when the state of an outgoing link changes, when a link ages-out, and periodically if the shortest-path tree has not been updated within a given time interval.

Data packets locally generated at the node waiting for a route to the destination are kept in the *data queue*. The packet at the tail of the queue is deleted when the queue is full and a new packet arrives to be enqueued. A packet is also deleted from the queue if a certain period of time has elapsed since it was inserted into the queue. A leaky bucket controls the rate with which data packets are dequeued for transmission in order to reduce the chances of congestion.

Each node maintains in the RREQ history table, for a specific length of time, a record with the source address and broadcast ID of each RREQ received. A node that receives a RREQ with a source address and broadcast ID already listed on the table does not forward the packet.

A node can send a RERR packet for a node *src*, through neighbor *nbr*, as a consequence of processing a data packet sent by node *src* to destination *dst*, after detecting the failure of the link (u, v), only if there is no entry in the RERR history table for the tuple (src, dst, u, v, nbr). An entry is deleted from the table after a certain period of time has elapsed since it was created. The RERR history table is a mechanism used to avoid the generation of a storm of RERR packets reporting the failure of the same link.

4.1.3 Routing Information Exchanged by NSR

NSR can generate four types of control packets: RREQ, RREP, RERR, and HELLO. Routing information is also sent in the header of data packets. RREQ and HELLO packets are broadcast unreliably and RREP and RERR are transmitted reliably as unicast packets. The link to the next-hop along the path to be traversed by RREP and RERR packets is taken down if after several retransmissions the link-layer fails to deliver the packet to the intended neighbor. All the packets transmitted by NSR have a field that keeps track of the number of hops traversed by the packets. A packet is not forwarded if the it has traveled MAX_PATHLEN hops, however, the link-state information it carries is processed.

RREP, RERR, and data packets contain a source route. The source route consists of the sequence of nodes to be traversed by the packet. The identification of a node in a source route does not need to be the node's address, it can be the *neighbor ID* assigned by the node that precedes it in the source route. The neighbor ID is encoded in 1 byte, representing a significant reduction in the overhead added by the source route in a data packet when the addresses of the nodes are encoded in several bytes (e.g., 4 bytes in IPv4 [44], or 16 bytes in IPv6 [9]).

Every packet but HELLOs is updated with the state of the link over which it was received (the receiving node is the head of the link and the neighbor which sent the packet is the tail of the link). The source of a data packet also adds to the source route the sequence number of the links along the path to be traversed. The receiver node processing a source route updates its topology graph with the state of the links traversed by the packet.

The link state information (LSI) reported by NSR for a given link consists of the cost the link (encoded in 1 byte), the sequence number of the head of the link (encoded in 2 bytes), and the lifetime of the link (encoded in 4 bits). LSIs reported in control packets have an extra field (encoded in 1 byte): the *neighbor ID* assigned by the head of the link to the tail node.

A node relaying RREQ, RREP, and RERR packets adds to the packet its *neighborhood link state* (NLS) which consists of the LSIs for outgoing links to neighbors. The NLS also contains the *partialLSI* flag which is set when the node reports a partial list of its outgoing links due to packet size constraints. All the links in an NLS have the same sequence number and lifetime. After processing the LSIs received in an NLS with the *partialLSI* flag not set, the node sets to infinity the cost of all the links in the topology graph having the same head node of the NLS but with a smaller sequence number, and the sequence number of these links is updated with the sequence number reported in the NLS. Consequently, the node that advertises its NLS does not have to report the set of links that were removed from its NLS due to failures.

HELLO packets carry the node's NLS and are not relayed by the receiving node. The receiver of a HELLO processes the NLS reported in the packet in the same way NLSs are processed when received in RREQ packets.

A RREQ packet contains the source node's address, the destination's address, the maximum number of hops it can traverse, and a broadcast ID which is incremented each time the source node initiates a RREQ (the broadcast ID and the address of the source node form a unique identifier for the RREQ). Two kinds of RREQs are sent: non-propagating RREQs which can travel at most one hop, and propagating RREQs which can be relayed by up to MAX_PATHLEN nodes.

A RERR packet is generated due to the failure of a link in the source route of a data packet. The RERR contains the source route received in the data packet, the head node of the failed link, the LSIs having as head node the head of the failed link, and the LSIs for the links in the alternate path to the destination (if any).

4.1.4 Operation of NSR

The NSR protocol is composed of four mechanisms that work together to allow the reliable computation of source routes on an on-demand basis:

- **Connectivity Management**: by which a node can learn the state of those links on the path to nodes two hops away. The cost of repairing a source route due to link failures can be significantly reduced by having available up-to-date state of such links.

- **Sequence Number Management**: by which the sequence number used in the validation of link-state information is updated such that its integrity is preserved across node-resets and network partitions. This mechanism also ensures that RREQs are uniquely identified across node-resets.

- **Route Discovery**: by which the source of a data packet obtains a source route to the destination when the node does not already know a route to it.
- **Route Maintenance**: by which any node relaying a data packet is able to detect and repair a source route that contains a broken link, and by which the source of a data packet is able to optimize source routes. A broken source route may be repaired multiple times until it reaches the destination without needing to notify the source of the data packet of such repairs.

Connectivity Management

This mechanism is responsible for determining the node's lifetime L_i and the state of the links to neighboring nodes. The link to a new neighbor is brought up when any packet is received from the neighbor. The link to a neighbor is taken down if the node does not receive any packet from the neighbor for a given period of time.

HELLO packets are broadcast periodically and have their transmission rescheduled when RREQ packets are transmitted.

The node's lifetime L_i is recomputed periodically based on the average time the links to neighbors are up. The minimum lifetime L_{min} is assumed to be 30 seconds and the maximum lifetime L_{max} is assumed to be 1800 seconds. When reported in an LSI the node's lifetime is encoded in 4 bits after being rounded down to the nearest of one of the following values (in seconds): L_{min}, 45, 60, 75, 90, 105, 120, 150, 165, 180, 240, 360, 480, 900, and L_{max}.

Sequence Number Management

NSR works with the assumption that only the source of data packets be notified of the failure of a link in the path being traversed by a data packet. Given that the cost of a link may change over time without being noticed by some nodes in the network that have the link in their topology graphs, link-state information must be aged-out to prevent routers from keeping stale routes. A node can ascertain whether the link-state information reported by a neighbor is valid by comparing the sequence number in the LSI against the sequence number stored in the topology graph for the same link. The router considers the received LSI as valid if its sequence number is greater than the sequence number stored for the same link, or if there is no entry for the link in the topology graph.

The sequence number used in the validation of link-state information consists of two counters maintained by the head node i of the link: the node's epoch E_i and the node's sequence number SN_i. Both E_i and SN_i are encoded in one byte each and have a value in the range [1, 254]. It is assumed that SN_i wraps around when its value is either 127 or 254 and it is incremented.

The value of SN_i is incremented whenever the router needs to advertise changes to its NLS. It is assumed that the time interval between node resets is greater or equal to $L_{min} = 30$ seconds, and that SN_i should wrap around in at least $L_{max} = 1800$ seconds, i.e., any node other than the head i of a link with lifetime set to L_{max} will have deleted the link from its topology graph by aging before E_i and SN_i wrap around. If SN_i wraps around before L_{max} seconds have elapsed since the previous wrap around, then E_i is incremented and SN_i is set to 1.

The procedure used to determine whether a value X based on SN_i or E_i is greater than a value Y is shown in Figure 4.2.

When SN_i gets incremented to a value sn greater than 127 then all the nodes in the network have already aged-out all the links reported by i with SN_i in the range [1, sn - 127]. Likewise, when SN_i gets incremented to a value sn smaller than 128 then all the routers have already aged-out all the links reported by i with SN_i in the range [128, sn + 127].

From the perspective of any node $x \neq i$ in the network, the combination of E_i and SN_i is seen as an unbounded counter because the values of E_i and SN_i have a lifetime.

The sequence number of a received LSI for the link (u, v) is greater then the sequence number stored in the topology graph for the same link if the E_u component of the LSI's sequence number is greater then the respective E_u stored in the topology graph, or if the epochs are the same but the SN_u component of the LSI's sequence number is greater than the respective SN_u component in the topology graph.

The broadcast ID set by node i in a RREQ packet also consists of two counters: the node's epoch E_i and a 4-byte sequence number B_i. The source of a RREQ increments B_i

```
result ← FALSE;

if ( X ≤ 127 )
  A ← X;   B ← Y;
else
  A ← Y;   B ← X;

if ( B ≤ 127 )
  if ( A > B )
    result ← TRUE;
else if ( | A - B | > 127 )
  result ← TRUE;

if ( X = A and result = TRUE )
  X is greater than Y;
else if ( Y = A and X ≠ Y and result = FALSE )
  X is greater than Y;
```

Figure 4.2: Procedure to determine if X is greater than Y, where X and Y are derived from E_i or SN_i

before creating the RREQ for transmission. By having E_i as part of the broadcast ID, RREQs are uniquely identified across node resets.

Route Discovery

When NSR receives a data packet from an upper-layer and the router has a source route to the destination, the source route is inserted into the packet's header and the packet is forwarded to the next hop towards the destination. Otherwise, NSR inserts the data packet into the *data queue* and initiates the *route discovery* process, if there is none already in progress, for the data packet's destination by broadcasting a non-propagating RREQ. By sending non-propagating RREQs, NSR prevents unnecessary flooding when the neighbors have a source route to the required destination. If none of the neighbors send a RREP within a timeout period, a propagating RREQ is transmitted. Each time a propagating RREQ is transmitted the timeout period is doubled until a pre-defined number of attempts have been made, after which it is kept constant. After a pre-defined number of RREQs have been transmitted for a given destination, the route discovery process is restarted by sending a non-propagating RREQ if the *data queue* holds a packet for the destination.

When a node receives a RREQ, it processes all the LSIs in the packet and then checks whether it has seen it before by comparing the source address and the broadcast ID from the RREQ against the entries in the RREQ history table. The RREQ is discarded if the node has already seen it before, otherwise the node is said to have received a *valid RREQ*, and an entry is added to the RREQ history table with the values of the RREQ's source address and broadcast ID. Non-propagating RREQs are always considered valid RREQs.

The receiver node of a non-propagating RREQ sends a RREP if it has a source route to the destination of the RREQ. Since a RREP to a non-propagating RREQ is not generated by the destination of the RREQ, the lifetime of the LSIs reported in the RREP must correspond

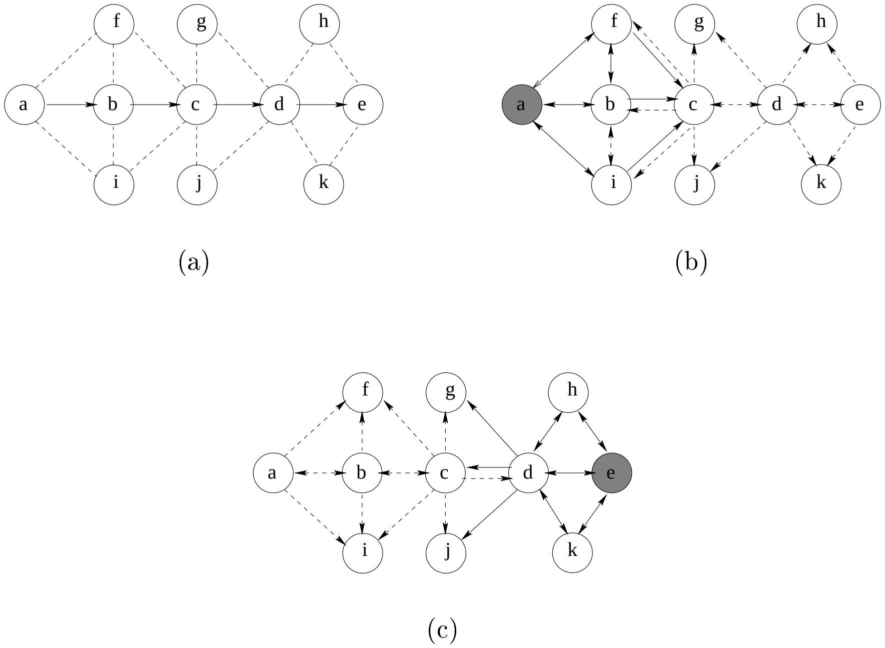
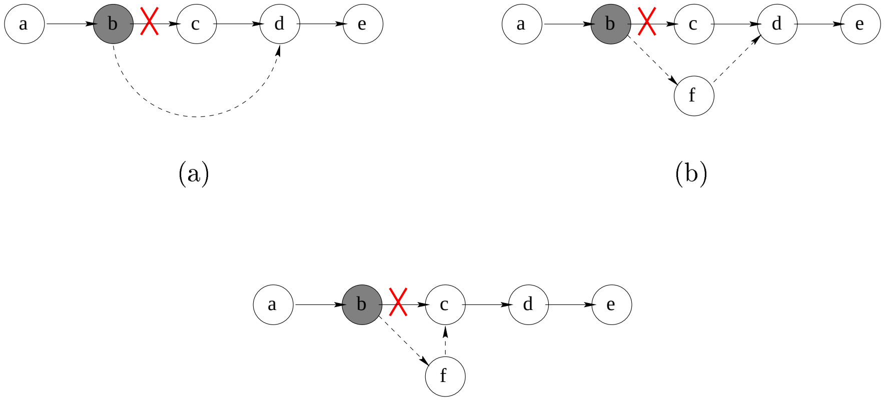


Figure 4.3: Link-state information learned from processing RREQ and RREP packets

to the time left for being aged-out from the node's topology graph.

If the node processing a valid RREQ is the destination of the RREQ then it sends a RREP back to the source of the RREQ. The source route contained in the RREP consists of the reversed path traversed by the RREQ packet. A node other than the destination of a valid RREQ adds its NLS into the packet before broadcasting it. Likewise, a node other than the destination of a RREP adds its NLS into the packet before forwarding it. The link-state information learned from RREQs and RREPs increases the chances of a node finding a source route in the topology graph and, consequently, increases the likelihood of replying to non-propagating RREQs. As an example, consider the network topology shown in Figure 4.3(a), where solid lines indicate the path traversed by the first RREQ packet received by destination node *e* from the source node *a*. The dashed lines in Figure 4.3(b) represent the links learned from the RREP received by node *a* from destination *e*. The dashed lines in Figure 4.3(c)



(a)

(b)

(c)

Figure 4.4: Type of repairs that can be applied to a source route in a RREP packet

represent the links learned from the RREQ received by node e from node a. The solid lines in Figures 4.3(b) and 4.3(c) correspond to those links learned from HELLO packets.

A node forwarding a RREP packet may change its source route if the link to the next hop has failed. The broken source route can be repaired if the node is able to find an alternate path having at most 2-hops to any of the nodes in the path to be traversed by the RREP packet. The order with which the nodes from the broken source route are visited when seeking an alternate path is from the tail node towards the node that corresponds to the tail of the failed link. Figure 4.4 shows the types of repairs that can be applied to the source route (shown in solid lines) in a RREP: in Figure 4.4(a) the failure of the link (b, c) causes node b to replace links (b, c) and (c, d) by (b, d), in Figure 4.4(b) the failure of the link (b, c) causes node b to replace links (b, c) and (c, d) by (b, f) and (f, d), and in Figure 4.4(c) the failure of the link (b, c) causes node b to replace link (b, c) by (b, f) and (f, c). An extra-hop can be added to a broken source route only if the length of the new path does not exceed MAX_PATHLEN hops.

Route Maintenance

A node forwarding a data packet attempts to repair the source route when either the link to the next hop or the link headed by the next hop in the path to be traversed has failed. The repair consists in finding an alternate path to the destination of the data packet, and may involve the transmission of a RERR packet to the source of the data packet.

The repair made by a forwarding node to the source route of a data packet does not trigger the transmission of a RERR packet if the following rules are satisfied:

- **Rule-1**: the node processing the packet is listed in the original source route received in the data packet.
- **Rule-2**: one of the nodes not yet visited by the data packet but listed in the original source route is at most two hops away from the router itself in the repaired source route.

The path traversed by a RERR packet consists of the reversed path traversed by the data packet, having as destination the source of the data packet that triggered its transmission. Rule-1 and Rule-2 guarantee that the source of the data packet is notified of all the link failures present in its source route. When the RERR reaches its destination, the source of the data packet updates its topology graph and recomputes its shortest-path tree.

Figure 4.5 illustrates the cases where repairs can be applied to the source route in a data packet without triggering the generation of RERRs. The links shown as solid lines in Figure 4.5 correspond to the source route carried by data packets originated at node a with destination f, and the dashed lines represent the links added to the new source route repaired by the nodes indicated with filled circles.

Figure 4.5(a) illustrates the fact that a node considers a source route as broken if any of the links in the next two hops following the node processing the packet has failed. In

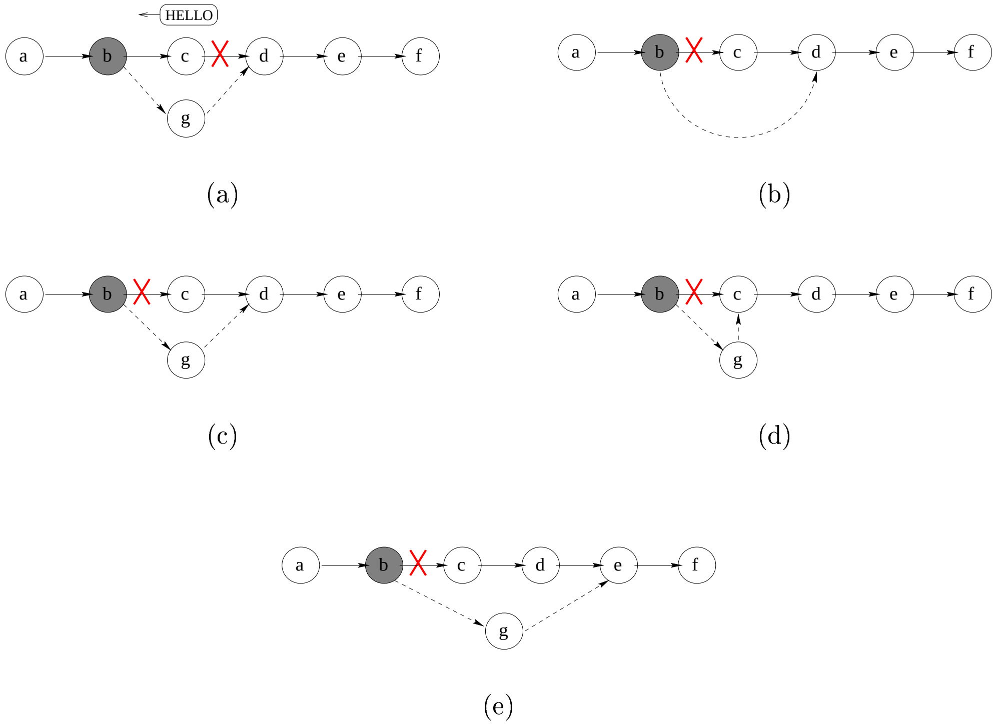


Figure 4.5: Type of repairs that can be applied to a source route in a DATA packet

this particular case, node b receives a HELLO packet from node c reporting the failure of link (c,d) before b receives a data packet to be forwarded. The node forwarding a data packet may not have in its topology graph the link that is one hop away in the source route. In order to prevent the node from dropping the data packet, NSR allows the packet to be forwarded if the sequence number in the source route for the missed link is greater then the sequence number of any link reported by the neighbor.

The failure of the link (b,c) shown in Figure 4.5(c) makes the links (b,c) and (c,d) in the source route be replaced by the links (b,g) and (g,d). The failure of the link (b,c) shown in Figure 4.5(d) causes the link (b,c) be replaced by the link (b,g) and the source route be extended in one hop by adding link (g,c) to it. The failure of the link (b,c) shown in

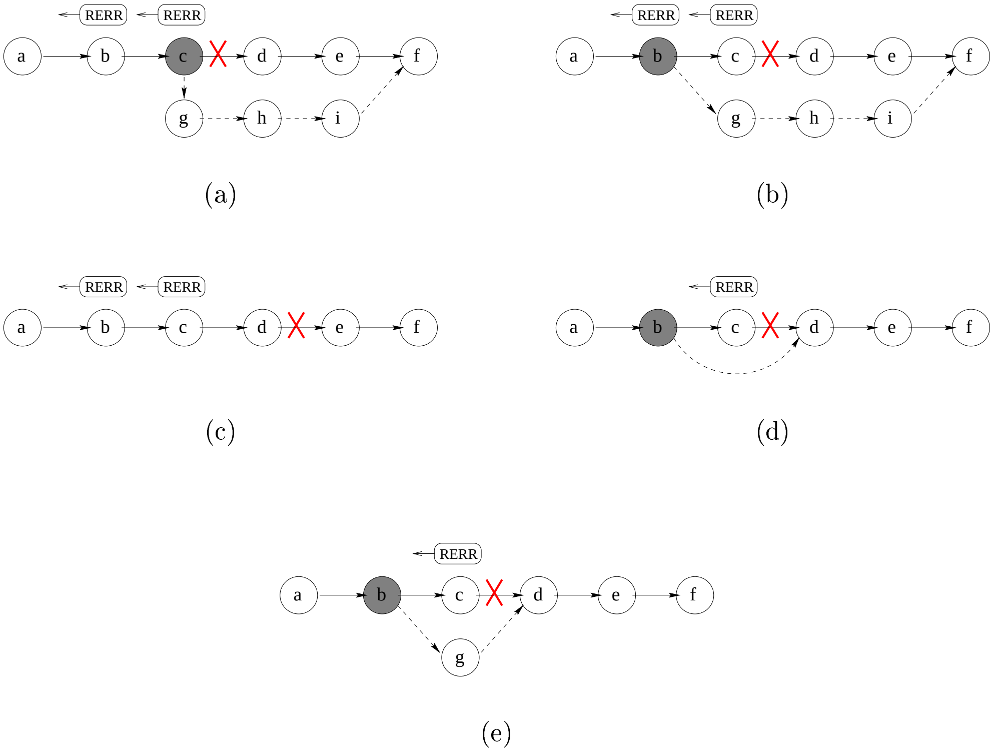


Figure 4.6: Broken source-routes leading to transmission of RERR packets

Figure 4.5(e) causes the link (b, c) be replaced by link (b, g) and the source route be shortened in one hop by replacing the links (c, d) and (d, e) by link (g, e).

If node g shown in Figure 4.5 receives a data packet with the source route repaired by node b and it detects the source route is broken, a RERR packet needs to be transmitted (even if g has an alternate path) since Rule-1 is not satisfied when the node attempts repairing the route.

Figure 4.6 illustrates the cases that trigger the transmission of a RERR packet when a source route in a data packet is detected to be broken. The links shown as solid lines in Figure 4.6 correspond to the source route carried by data packets originated at node a with destination f, and the dashed lines represent the links added to the new source route repaired by the nodes indicated with filled circles.

The generation of RERR packets reporting the failure of the same link are spaced by some time interval if the source route being processed was generated by the same source node, and the data packet is for the same destination, and the data packet was received from the same neighbor that caused the transmission of the previous RERR packet. This mechanism prevents the generation of a RERR packet for every data packet in transit carrying the same source route. After transmitting a RERR packet the node updates its RERR history table by adding an entry with information about the RERR packet.

The failure of the link (c, d) shown in Figure 4.6(a) causes node c to transmit a RERR packet to the source of a data packet received for forwarding. The RERR packet reports the new source route to destination f, which consists of the links (c, g), (g, h), (h, i), and (i, f) instead of the links (c, d), (d, e), and (e, f). The data packet being processed has its source route updated accordingly and is forwarded to node g. When node b receives the RERR packet its topology graph is updated with the link-state information reported in the packet, its shortest-path tree is recomputed, its NLS is added to the packet, and the packet is then forwarded to a.

Node b shown in Figure 4.6(b) adds to the RERR packet received from node c an alternate path to f before forwarding the packet to a. NSR allows the node forwarding a RERR to add an alternate path to the RERR packet only if it is a neighbor of the head node of the failed link that triggered the generation of the RERR packet.

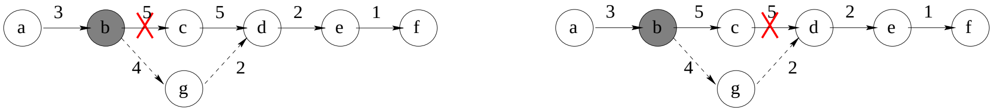
Node a shown in Figure 4.6(c) receives a RERR packet not reporting an alternate path to the destination. Node b shown in Figures 4.6(d) and 4.6(e) does not forward the RERR packet because it has an alternate path to the destination. The next data packet it receives from a has the source route repaired with the alternate path.

4.1.5 Using Neighbor IDs in Source Routes

The source route given in a data packet can be formed by the sequence of neighbor IDs mapped to each link along the path to be traversed by the packet, instead of being formed by node's addresses. Such approach makes the source route very compact and allows more data be carried in each packet. As an example consider the scenarios depicted in Figure 4.7.

The numbers beside links shown as solid lines in Figure 4.7 correspond to the source route carried by data packets originated at node a with destination f, and the dashed lines represent the links added to the new source route repaired by the nodes indicated with filled circles. The number beside a link is the neighbor ID given by the head of the link to the tail node. The *neighbor table* contains the mapping between neighbor ID and the address of a neighbor. The entry for the node with neighbor ID 5 is not deleted from the neighbor table of node b when the link (b, c) fails (Figure 4.7(a)), allowing b to get the address of c and repair the source route accordingly. Because links are deleted from the topology graph only due to aging, node b in Figure 4.7(b) is able to identify the tail of the failed link (c, d) by looking for all the links in the topology graph having node c as the head of the link and a neighbor ID 5.

The source route received by node f in the data packet sourced at a contains the state of all the links in the reversed path traversed by the data packet. With the failure of either link (b, c) or link (c, d) the source route received by node f contains LSIs for the links (d, g) and (g, b). Node f cannot update the topology graph with the state of (d, g) and (g, b) if the links



(a) (b)

Figure 4.7: Using *neighbor IDs* in source routes

are not part of f's topology graph and the source route lists neighbor IDs instead of node's addresses. The likelihood of finding alternate paths to destinations increases with up-to-date link-state information carried in data packets, specially when the data flows are bidirectional. For this reason, the source of data packets are required to periodically use addresses instead of neighbor IDs in the source routes.

4.2 Proof of Correctness

In what follows, we show that NSR works correctly by showing that NSR discovers source routes to any destination in a connected network within a finite time, that the source of data packets is notified within a finite time when the destination becomes unreachable, that data packets cannot form any routing loop, and that NSR achieves correct reset of sequence numbers.

For simplicity we assume that the link-layer can inform NSR about link failure within a finite time after the link fails. We also assume that the rate of changes in link connectivity a router can experience does not exceed C_{max} = 15 changes per second. This assumption is realistic given the fact that so many changes in link connectivity would cause routing instability in a network and only broadcast of data packets would be feasible. It is assumed that NSR starts its operation at least W_{min} = 30 seconds after the router resets. It is also assumed that the propagation time of any packet from the source to the destination does not exceed P_{max} = 300 seconds. The value of MAX_PATHLEN is a finite number used by NSR to determine the maximum number of hops any packet can traverse in the network.

Theorem 15 *NSR achieves correct reset of sequence numbers.*

Proof: The sequence number reported by router i in the LSIs for its outgoing links consists of the counters E_i and SN_i. Both E_i and SN_i have a value in the range [1, 254], and the value of E_i is stored in non-volatile storage when it is incremented. Assume that $E_{max} = SN_{max} = 254$ is the maximum value E_i and SN_i can have. The value of SN_i wraps around either when it is incremented from 127 to 128 or when it changes from 254 to 1. When SN_i wraps around before L_{max} (maximum lifetime of an LSI) seconds have elapsed since the previous wrap around, E_i is incremented. E_i is also incremented after the router resets and before NSR starts its operation. Assuming that a control packet needs to be transmitted whenever a change in link connectivity occurs, router i may have to increment its sequence number C_{max} times in 1 second. Thus, given that at most $LC = C_{max} * L_{max}$ changes in link connectivity can occur in L_{max} seconds, and that $LC < (E_{max} - 1) * (SN_{max}/2)$, and that $((E_{max} - 1) * (SN_{max}/2) - LC)/C_{max} > P_{max}$ it is clear that it takes more than L_max seconds between wraparounds of E_i. The Theorem is true because the time elapsed between wraparounds of E_i is greater than L_{max} seconds, the maximum lifetime for an LSI, and because node resets can cause E_i to be incremented at most $L_{max}/W_{min} < E_{max}$ times in L_{max} seconds. $\square$

Theorem 16 *The path traversed by RREQ packets is finite and has no cycles.*

Proof: The maximum number of hops a RREQ can traverse is given in the packet's header ($RREQ_{maxHops}$) by the source of the RREQ. After decrementing by one the value of $RREQ_{maxHops}$, the receiver node of a RREQ forwards the packet if $RREQ_{maxHops} \neq 0$ and the node is not the destination of the RREQ. The Theorem is true for a non-propagating RREQ because the source of the packet initializes $RREQ_{maxHops}$ to 1. The path traversed by a propagating RREQ is recorded in the packet. NSR's operation is such that a node discards a propagating RREQ if the node is listed in the path traversed by the packet, thus preventing the formation of a cycle. The Theorem is proved because the path traversed by a propagating RREQ does not have a cycle and the value of $RREQ_{maxHops}$ is initialized to MAX_PATHLEN by the source of the RREQ. □

Theorem 17 *The path traversed by RREP packets is finite and has no cycles.*

Proof: The source route added to a RREP by the source of the packet consists of the reversed path traversed by a RREQ packet. Due to Theorem 16 this source route is finite and has no cycle. NSR's operation is such that when a forwarding node finds an alternate path for a broken source route 1) the path to be traversed in the new source route does not contain any of the nodes from the traversed path, 2) the path traversed by the RREP packet is included in the new source route, 3) and the total number of nodes in the new source route cannot exceed MAX_PATHLEN. Therefore, the source route repaired by a forwarding node is of finite length with no cycles. Given that forwarding nodes discard RREP packets that cannot have the source route repaired and that the destination of a RREP packet does not forward the packet, then the Theorem is proved because neither the source of the packet nor the forwarding nodes introduce a cycle in the source route and the source route is always of finite length. □

Theorem 18 *The path traversed by data packets is finite and has no cycles.*

Proof: The source routes maintained by a node are obtained by running the Dijkstra's shortest-path tree algorithm on the topology graph. Because the cost of all the links in the topology graph is positive, the source route for any destination in the shortest-path tree consists of a path with no cycles. Because NSR computes paths having at most MAX_PATHLEN nodes the source route for any destination in the shortest-path tree consists of a finite number of nodes. NSR's operation is such that when a forwarding node finds an alternate path for a broken source route 1) the path to be traversed in the new source route does not contain any of the nodes from the traversed path, 2) the path traversed by the data packet is included in the new source route, 3) and the total number of nodes in the new source route cannot exceed MAX_PATHLEN. Therefore, the source route repaired by a forwarding node is of finite length with no cycles. Given that forwarding nodes discard data packets that cannot have the source route repaired and that the destination of a data packet does not forward the packet, then the Theorem is proved because neither the source of the packet nor the forwarding nodes introduce a cycle in the source route and the source route is always of finite length. $\Box$

Theorem 19 *The path traversed by RERR packets is finite and has no cycles.*

Proof: The source route added to a RERR by the source of the packet consists of the reversed path traversed by a DATA packet. Due to Theorem 18 this source route is finite and has no cycle, then the Theorem is proved because forwarding nodes cannot change the source route of a RERR packet. $\Box$

Lemma 5 *If at time t_0 a node s chooses a source route for destination d and the source route is not correct, then s will stop using the source route within a finite time.*

Proof: Suppose that node s chooses a source route for d and within a finite time starts transmitting data packets using that source route. A forwarding node sends a RERR packet if it does not have a correct source route to any of the nodes in the path to be traversed by the received data packet. A source route is considered correct if its concatenation with the path traversed by the data packet results in a path with no cycles and of finite length. When data packets are forwarded along a source route, they will either reach a node s_n, at a distance of n hops from s, which either has a correct source route to d or has no correct source route to d. If the first condition is satisfied then the Lemma is proved because data packets can be routed towards d based on s's source route to d. For the second condition the problem at s_n becomes similar to the problem at s, which implies a recursion. Because a source routes is finite, after the data packet has traversed a finite number of hops, a node s_n either selects a node s_{n+1} with no correct source route to any of the nodes in the path to be traversed or the node itself has no correct source route to any of the nodes in the path to be traversed. In the first case, s_{n+1} sends a RERR advertising a higher sequence number for the head node of the failed link, which s_n still thinks exists. Node s_n processes the RERR within a finite time updating its source routes, and forwards the RERR packet to s_{n-1} if it does not have a correct source route to any of the nodes in the path to be traversed by the data packet previously sent to s_{n+1}. Node s_{n-1} forwards the RERR packet received from s_n reporting the link failure regardless of having an alternate path to the destination. Accordingly, within a finite time node s will update its source routes and data packets stop flowing along the source route with the failed link.

RREQ packets are also sent when a forwarding node finds a correct source route to the destination but fails to find a correct source route having any of the nodes in the path to be traversed at most two hops away from itself. With the alternate path received in such RERR packets the source of data packets starts using a new source route that has a better chance of being repaired by a forwarding node without triggering the transmission of new RREQs. □

Lemma 6 *If a node does not have a source route to a destination d and has a data packet for d, it obtains a correct source route to d, if there exists any, within a finite time after sending a RREQ.*

Proof: A node s initiates a route discovery process by sending a non-propagating RREQ. If none of the neighbors has a source route, a propagating RREQ is sent which traverses multiple hops. Node s sends propagating RREQs periodically, in finite time intervals, while the node has a data packet for d and a RREP has not been received. Destination d should be able to receive at least one of the propagating RREQs within a finite time because the network is of finite size and is connected. By Theorem 17 the RREP sent by d or by a neighbor should arrive at node s within a finite time of its transmission because the RREP's source route is finite and has no cycles. The Lemma is true because the source route learned from a RREP packet is of finite length and has no cycles.

The proof is still valid if a node sends in its RREP an old path. This will make the data packet flow along the wrong path for some time, but due to Lemma 5, the error would be detected within a finite time and in the worst case another RREQ has to be sent. □

Theorem 20 *If a node becomes disconnected at time t_1, every node with a source route to the node at $t_0 < t_1$, will have no path to it within a finite time.*

Proof: Each node failure can be assumed to be equivalent to multiple link failures. Therefore, using Lemmas 6 and 5 we can say that after a node failure every node wishing to reach the failed node will have no path to it. □

4.3 Performance Evaluation

We run a number of simulation experiments to compare the average performance of NSR with respect to STAR and DSR. Both NSR, STAR, and DSR use the services of a medium access (MAC) protocol based on an RTS-CTS-DATA-ACK packet exchange for unicast traffic (similar to the IEEE 802.11 standard [7]). The promiscuous mode of operation is disabled on DSR because the MAC protocol uses multiple channels to transmit data. (Both NSR and DSR might benefit from having the node's network interface running in promiscuous mode.) The physical layer is modeled as a direct sequence spread spectrum radio with a link bandwidth of 1 Mbit/sec, accurately simulating the physical aspects of a wireless multi-hop network.

In order to show that NSR scales better than STAR we use a small network formed by 20 routers. The simulation experiments not involving STAR are based on a 50-node network.

In the simulation experiments described in this Section, a router running NSR and relaying RREQ, RREP, and RERR packets adds to the packet its *neighborhood link state* (NLS) and there is no transmission of HELLO packets. Two variants of NSR are simulated in order to determine the gains introduced by different mechanisms:

- **NSR-HE**: Unlike **NSR**, HELLO packets reporting the node's NLS are transmitted periodically.
- **NSR-NL**: Unlike **NSR-HE**, a node relaying RREQ, RREP, and RERR packets replaces the NLS information added by the sender of the packet with its own NLS.

The two variants of DSR are simulated and analyzed: the first utilizes a cache data structure based on complete paths (path cache) and is referred simply by **DSR**, the second (**DSR-LC**) has a cache data structure based on a graph representation of individual links (link cache).

4.3.1 Protocol Configuration

The values for the constants controlling DSR operation during the simulations are those present in the *ns-2* implementation of DSR [15].

Since the performance of adaptive routing caches is comparable to that of well-tuned static caches, we chose to use the *Link-Static-5* [27] link cache algorithm in our simulated experiments. This is a generational cache, such that the source of data packets marks the links used in the source route to not timeout. A link is deleted from the routing cache of a node if the source routes used to route the node's data packets have not included the link in a 5 seconds interval.

Routers running STAR are configured to send incremental updates and exchange update messages according to the rules LORA-1 to LORA-3.

The values for the constants controlling NSR operation are listed below:

- Time between successive transmissions of RREQs for the same destination is 0.5 seconds. This time is doubled with each transmission and is kept constant to 10 seconds with the transmission of the sixth RREQ.
- The minimum lifetime of an LSI is 30 seconds, and the maximum lifetime is 1800 seconds.
- The average time interval between the transmission of HELLO packets is 59 seconds with a standard deviation of 1 second (used by the NSR-HE and NSR-NL variants).
- The maximum number of entries in the *data queue*, RREQ history table, and RERR history table is 50, 200, and 200, respectively.
- The lifetime of an entry in the *data queue*, RREQ history table, and RERR history table is 30, 30, and 5 seconds, respectively.
- Data packets with a source route are removed from the *data queue* for transmission spaced from each other by 50 milliseconds.
- The maximum number of nodes (MAX_PATHLEN) traversed by any packet is 10.

4.3.2 Comparison with STAR

The protocol evaluations are based on the simulation of 20 wireless nodes in continuous motion (*pause time* 0) for 900 seconds of simulated time. These experiments use the same data traffic model presented in Section 3.4.4.

Tables 4.1 and 4.2 summarize the behavior of NSR, STAR, and DSR. The Tables show the total number of update packets transmitted by the nodes and the percentage of data packets delivered to the applications for the three simulated workloads.

Flows	NSR	NSR-HE	NSR-NL	STAR	DSR	DSR-LC
8	181	319	340	585	791	617
14	159	336	356	560	1251	564
20	328	325	398	575	3298	852

Table 4.1: Number of control packets generated by NSR, STAR, and DSR in a 20-node network

Flows	NSR	NSR-HE	NSR-NL	STAR	DSR	DSR-LC
8	0.67	0.64	0.65	0.62	0.61	0.69
14	0.62	0.55	0.56	0.63	0.63	0.53
20	0.59	0.64	0.65	0.58	0.23	0.56

Table 4.2: Percentage of data packets delivered by NSR, STAR, and DSR in a 20-node network

STAR generates 2.5 times more control packets than NSR when data packets are destined to a subset of the routers in the network. NSR is more bandwidth efficient than STAR even when all the destinations in the network are sink of data packets, i.e., NSR transmits 57% of the number of control packets sent by STAR.

While DSR-LC shows to be more efficient than DSR both in bandwidth usage and delivery of data packets, it generates more control packets than STAR when data packets are destined to all the nodes in the network. NSR delivers on average more data packets than both STAR and DSR. Overall, NSR presents the best performance regardless of the data traffic pattern inserted in the network.

4.3.3 Comparison with DSR

Mobility Pattern

The simulation experiments use 50 nodes moving over a rectangular flat space of 5Km x 7Km and initially randomly distributed at a density of 1.5 nodes per square kilometer. Nodes move in the simulation according to the *random waypoint* model [11]. In this model, each node begins the simulation by remaining stationary for *pause time* seconds, it then selects a random destination and moves to that destination at a speed of 20 meters per second for a period of time uniformly distributed between 5 and 11 seconds. Upon reaching the destination, the node pauses again for *pause time* seconds, selects another destination, and proceeds there as previously described, repeating this behavior for the duration of the simulation.

The simulation experiments are run for the pause times of 0, 15, 30, 45, 60, 90, and 900 seconds, and the total simulated time in all the experiments is 900 seconds. A pause time of 0 seconds correspond to the continuous motion of the nodes, and in a pause time of 900 seconds the nodes are stationary.

Data Traffic Model

The overall goal of the simulation experiments is to measure the ability of the routing protocols to react to changes in the network topology while delivering data packets to their destinations. The aggregate traffic load generated by all the flows in a simulated network consists of 32 packets/second and the size of a data packet is 64 bytes. The data traffic load was kept small to ensure that congestion of links is due only to heavy control traffic.

We applied to the simulated network three different communication patterns: a pattern of N-sources and N-destinations (Nsrc-Ndst), a pattern of N-sources and 8-destinations (Nsrc-8dst), and a pattern of N-sources and 1-destination (Nsrc-1dst). For each communication pattern we run a number of simulation experiments with different number of data flows. The data flows consist of continuous bit rate traffic, all the flows in a simulation experiment generate traffic at the same data rate, and each node is the source of no more than one data flow. We run four simulation experiments with 8, 16, 32, and 50 sources for both the Nsrc-Ndst and the Nsrc-1dst patterns, and 3 simulation experiments with 16, 32, and 50 sources for the Nsrc-8dst pattern. The data flows are started at times uniformly distributed between 10 and 120 seconds of simulated time.

Comparison Between Different Variants of NSR

Figures 4.8, 4.9, 4.10, and 4.11 summarize the comparative performance of the different variants of NSR in all the simulation experiments. The behavior of the protocols according to the node mobility and traffic pattern is depicted in more detail in Figures 4.17 to 4.22. (The coordinates having a value of 100 in the x-axis correspond to the results obtained for networks with stationary nodes.)

As shown in Figure 4.8, in most of the simulated experiments NSR generated significantly less control packets than NSR-HE and NSR-NL. In 40% of the experiments NSR sent less than 1300 control packets, while NSR-HE and NSR-NL produced that same amount of control packets in 12% and 9% of the experiments, respectively. The extra link-state information reported in RREQ, RREP, and RERR packets enabled routers running NSR to find alternate paths to broken source routes without starting a Route Discovery process and, consequently, sending less control packets than NSR-NL.

NSR-HE and NSR-NL were able to deliver more data packets than NSR (Figure 4.9) in the NsrcNdst data pattern. NSR-HE and NSR-NL need to send control packets more frequently than NSR to repair a broken source route. Since RREQ packets propagate faster towards the destination along less congested links, the new paths resulting from a Route Discovery process usually tend to deliver more data packets to the destination than the source routes repaired with the extra link-state information maintained in the topology graph. This explains why NSR-HE and NSR-NL delivered more data packets than NSR under the NsrcNdst data pattern, at the expense of generating more control packets when compared to the other data patterns. Both the propagation delay experienced by data packets and the number of hops traversed by data packets is about the same among the three variants of NSR.

Overall, NSR showed to be more efficient than its two variants NSR-HE and NSR-NL.

Comparison Between NSR and Variants of DSR

Figures 4.12, 4.13, 4.14, 4.15, and 4.16 summarize the comparative performance of NSR and DSR in all the simulation experiments. The behavior of the protocols according to the node mobility and traffic pattern is depicted in more detail in Figures 4.23 to 4.28. (The coordinates having a value of 100 in the x-axis correspond to the results obtained for networks with stationary nodes.)

As shown in Figure 4.12, in most of the simulated experiments NSR generated significantly less control packets than DSR-LC and DSR. From Figure 4.13 we can see that DSR-LC and DSR are able to generate less than 4000 control packets in only 47% and 19% of the experiments, respectively, while NSR produces that amount in 91% of the experiments. The maximum number of control packets transmitted by NSR, DSR-LC, and DSR in a simulated experiment was 7330, 16260, and 83010, respectively.

Figures 4.23, 4.24, and 4.25 show that the type of workload introduced in the network makes DSR to behave in an unpredictable manner while NSR and DSR-LC are very little affected. In the simulation experiments with Nsrc-Ndst traffic pattern DSR generates up to 9.9 times more control packets than NSR, with Nsrc-1dst traffic pattern DSR generates up to 45 times more control packets, and with Nsrc-8dst traffic pattern DSR generates up to 18.5 times more control packets than NSR.

The benefits of using link-state information instead of path information for routing decisions is noticeable when N sources send data packets to the same destination (Nsrc-1dst traffic pattern): most of the ROUTE REPLIES generated by nodes running DSR carry stale routing information, which leads to the transmission of more ROUTE REQUESTS. And, as shown in Figure 4.24, this behavior is independent of the *pause time* (except when the nodes are stationary and the number of data flows is low).

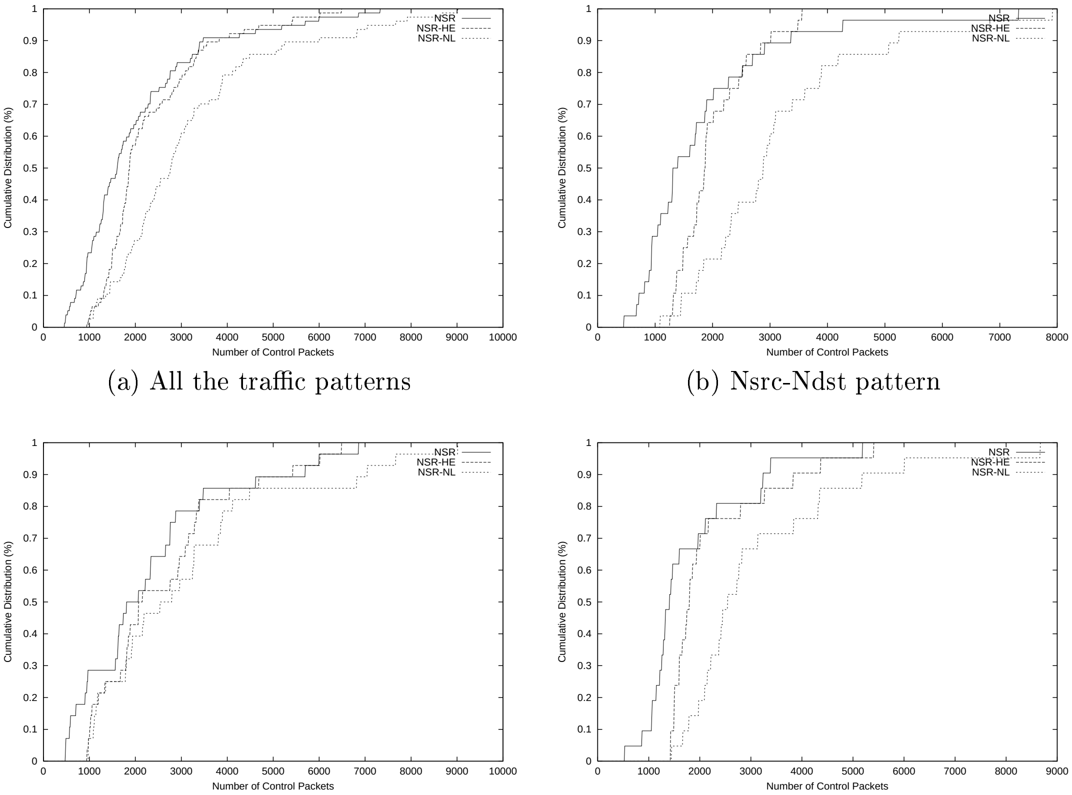
Figures 4.26, 4.27, and 4.28 give the average performance of NSR, DSR-LC, and DSR in terms of the percentage of data packets delivered to the destinations for a given *pause time* and traffic pattern. We see that the highest number of packets are delivered as the nodes become less mobile. This behavior is expected because all the packets enqueued for transmission at the link-layer are dropped after link failures, and link failures occur less frequently when the nodes in the network become more stationary. Because DSR-LC needs to perform Route Discovery much more frequently than NSR, under high node mobility (pause time 0) NSR delivered less data packets due to congested links. Overall, NSR delivers about the same number of data packets than DSR-LC.

We observe from Figure 4.14(a) that, on average, NSR is able to deliver to the destinations more data packets than DSR. The lack of a source route to the destination in the experiments with Nsrc-1dst traffic pattern caused DSR to discard a high number of packets awaiting for a route (Figure 4.14(c)).

The end-to-end delay experienced by data packets (Figure 4.15) routed by NSR is slightly higher than the end-to-end delay experienced by data packets routed by DSR-LC and DSR because routers running NSR tend to share the same paths towards a destination, particularly when source routes are repaired by forwarding routers. This behavior is clearly noticeable when all the routers send data to the same destination (Figure 4.15(c)).

We observe from Figure 4.16 that the number of hops traversed by data packets routed by NSR and DSR-LC is about the same, while DSR makes use of longer paths. Most of the data packets routed by NSR and DSR-LC traversed from 2 to 3 hops, while most of the packets routed by DSR traversed from 2 to 4 hops.

Overall, NSR outperforms both DSR-LC and DSR.



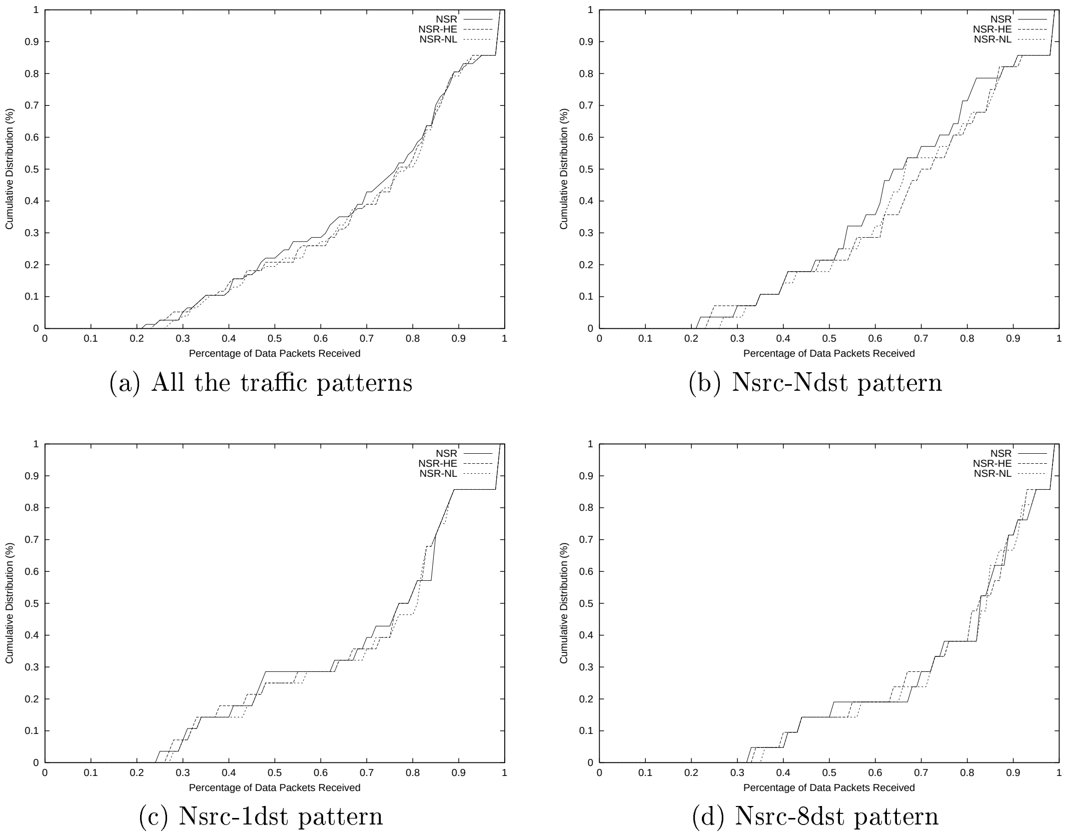
(a) All the traffic patterns

(b) Nsrc-Ndst pattern

(c) Nsrc-1dst pattern

(d) Nsrc-8dst pattern

Figure 4.8: Cumulative distribution function for the number of control packets generated



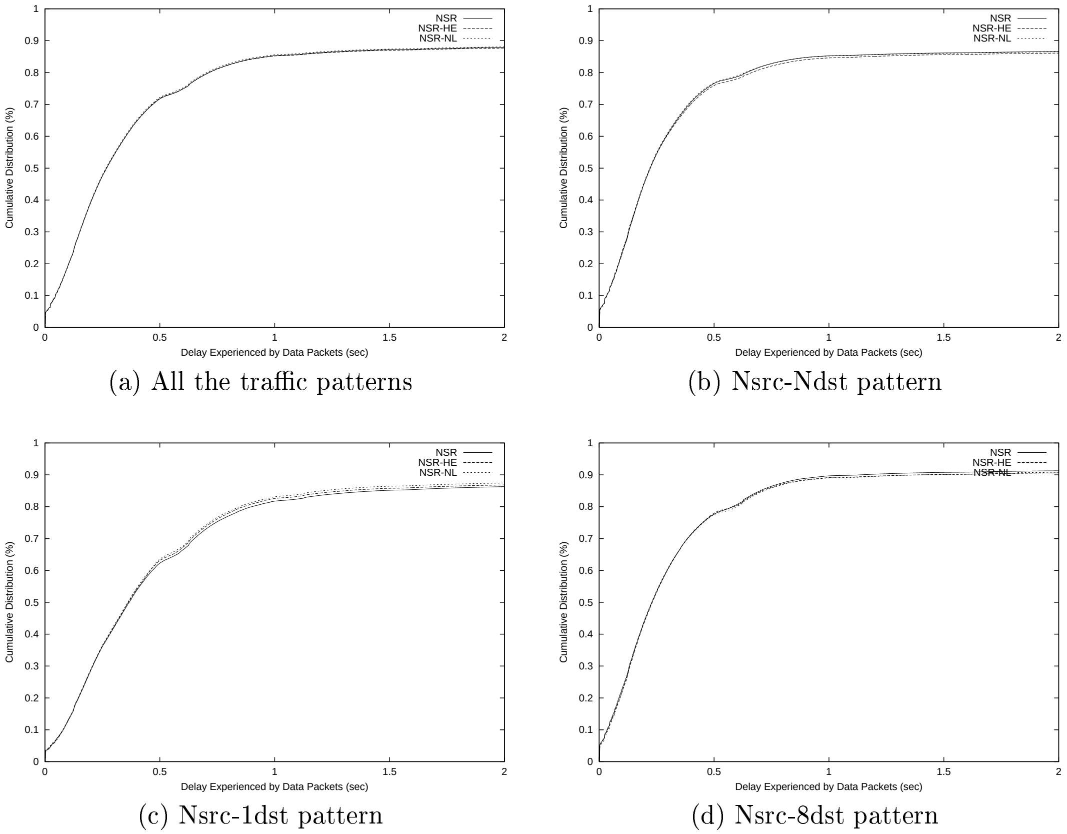
(a) All the traffic patterns

(b) Nsrc-Ndst pattern

(c) Nsrc-1dst pattern

(d) Nsrc-8dst pattern

Figure 4.9: The cumulative distribution function for the percentage of data packets received



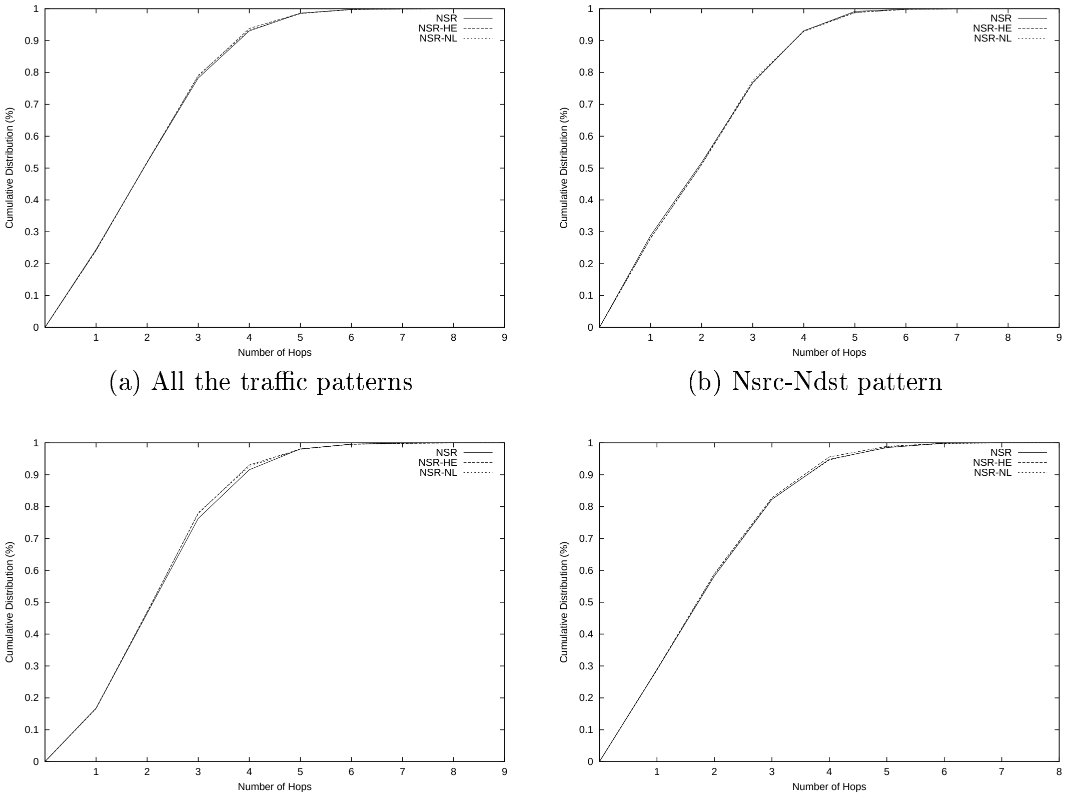
(a) All the traffic patterns

(b) Nsrc-Ndst pattern

(c) Nsrc-1dst pattern

(d) Nsrc-8dst pattern

Figure 4.10: The cumulative distribution function for the delay experienced by data packets



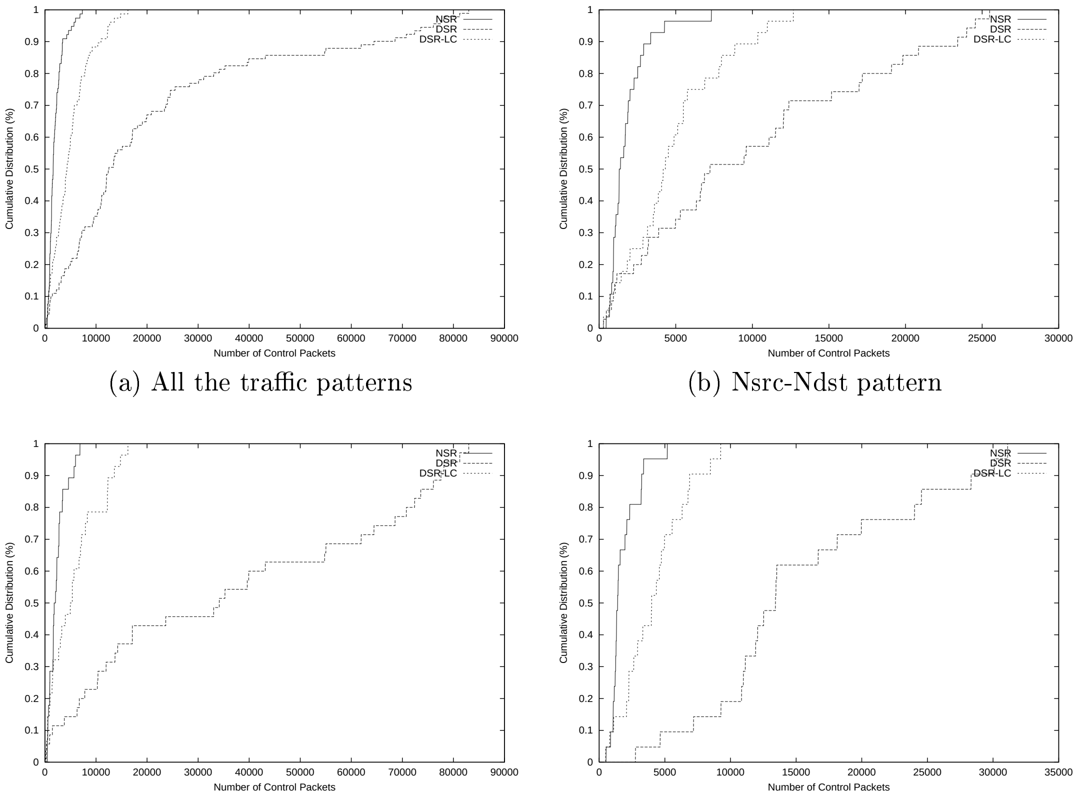
(a) All the traffic patterns

(b) Nsrc-Ndst pattern

(c) Nsrc-1dst pattern

(d) Nsrc-8dst pattern

Figure 4.11: The cumulative distribution function for the number of hops traversed by data packets



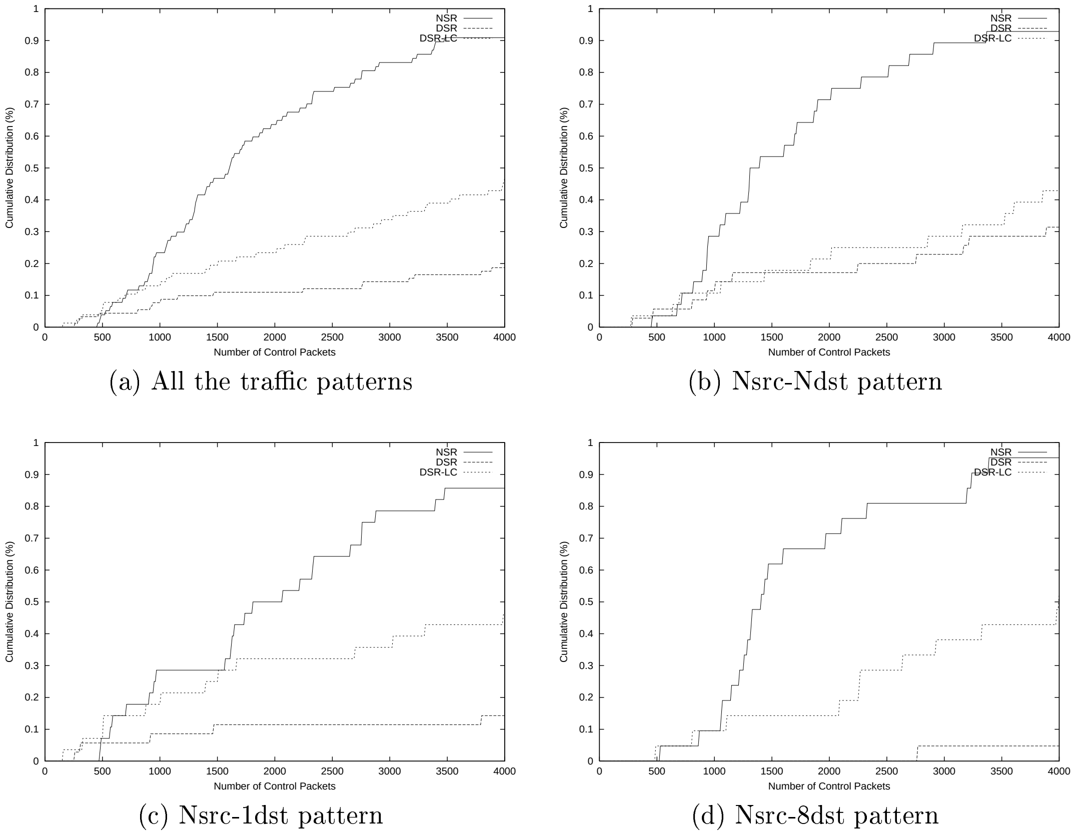
(a) All the traffic patterns

(b) Nsrc-Ndst pattern

(c) Nsrc-1dst pattern

(d) Nsrc-8dst pattern

Figure 4.12: Cumulative distribution function for the number of control packets generated



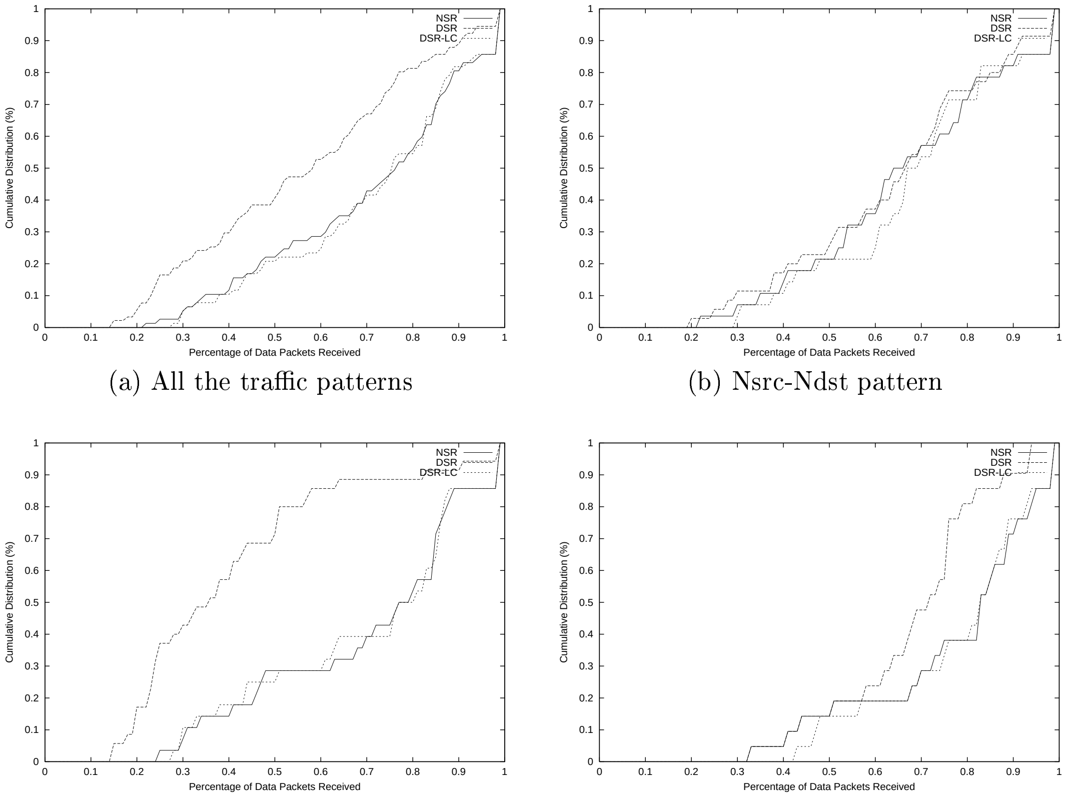
(a) All the traffic patterns

(b) Nsrc-Ndst pattern

(c) Nsrc-1dst pattern

(d) Nsrc-8dst pattern

Figure 4.13: A partial view of the cumulative distribution function for the number of control packets generated



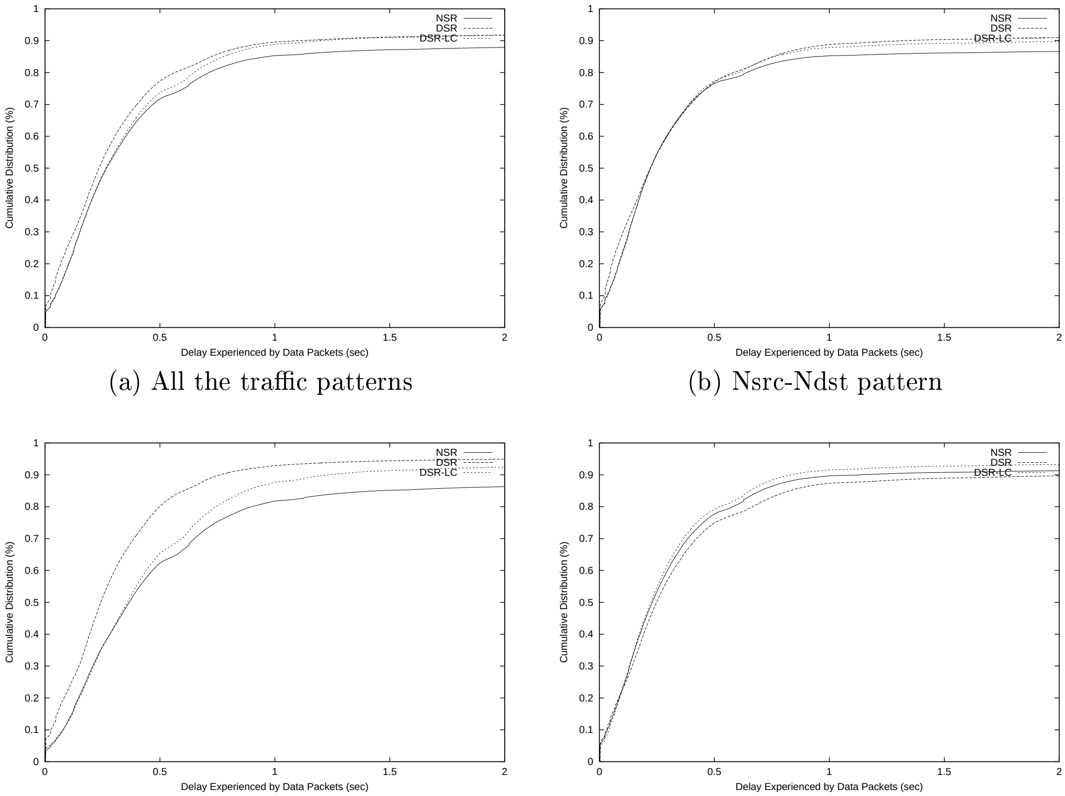
(a) All the traffic patterns

(b) Nsrc-Ndst pattern

(c) Nsrc-1dst pattern

(d) Nsrc-8dst pattern

Figure 4.14: The cumulative distribution function for the percentage of data packets received



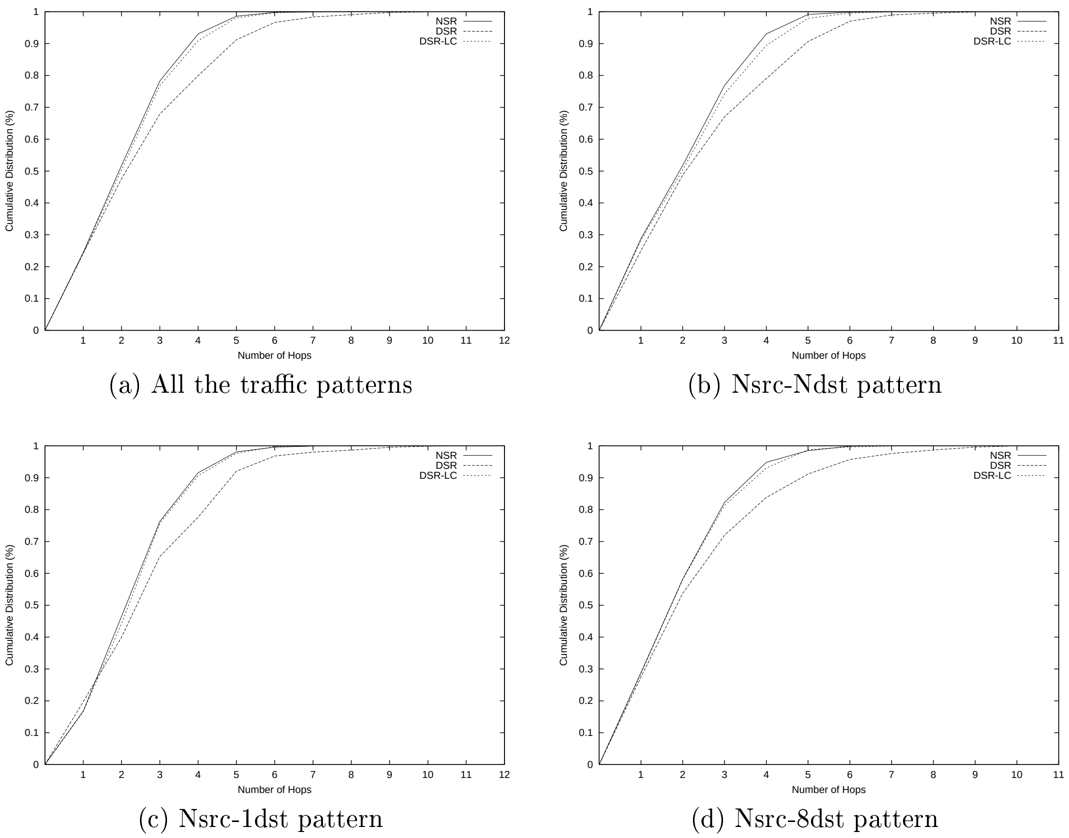
(a) All the traffic patterns

(b) Nsrc-Ndst pattern

(c) Nsrc-1dst pattern

(d) Nsrc-8dst pattern

Figure 4.15: The cumulative distribution function for the delay experienced by data packets



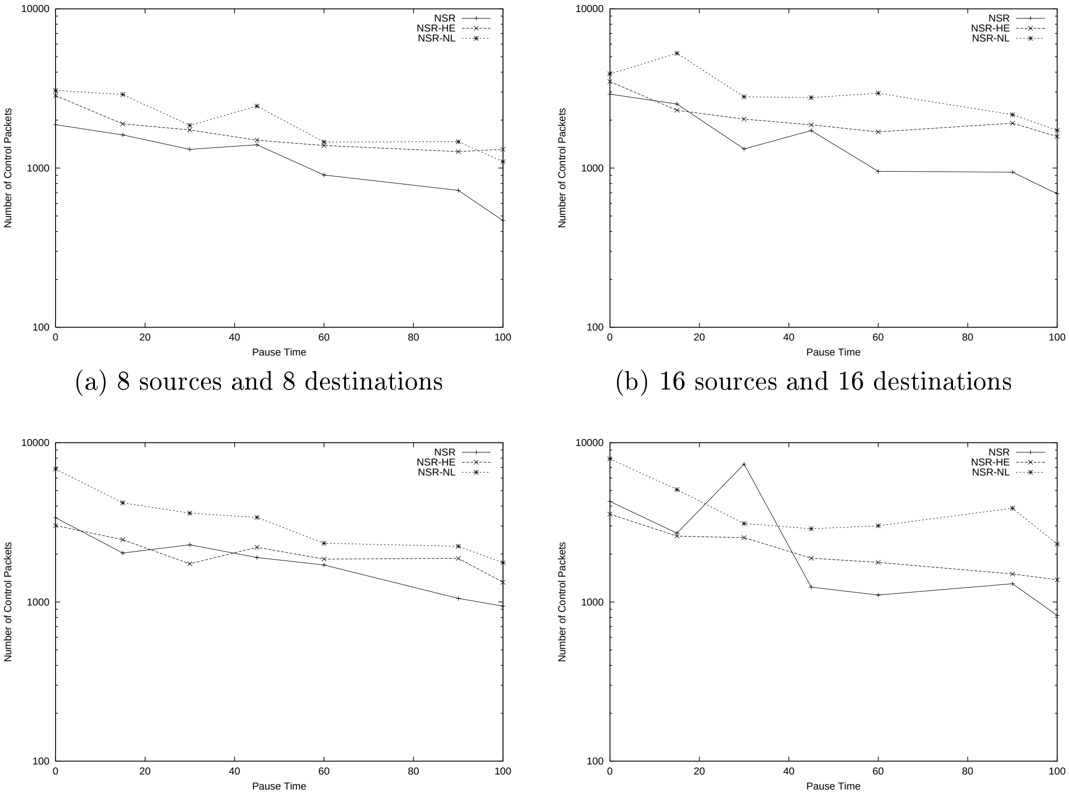
(a) All the traffic patterns

(b) Nsrc-Ndst pattern

(c) Nsrc-1dst pattern

(d) Nsrc-8dst pattern

Figure 4.16: The cumulative distribution function for the number of hops traversed by data packets



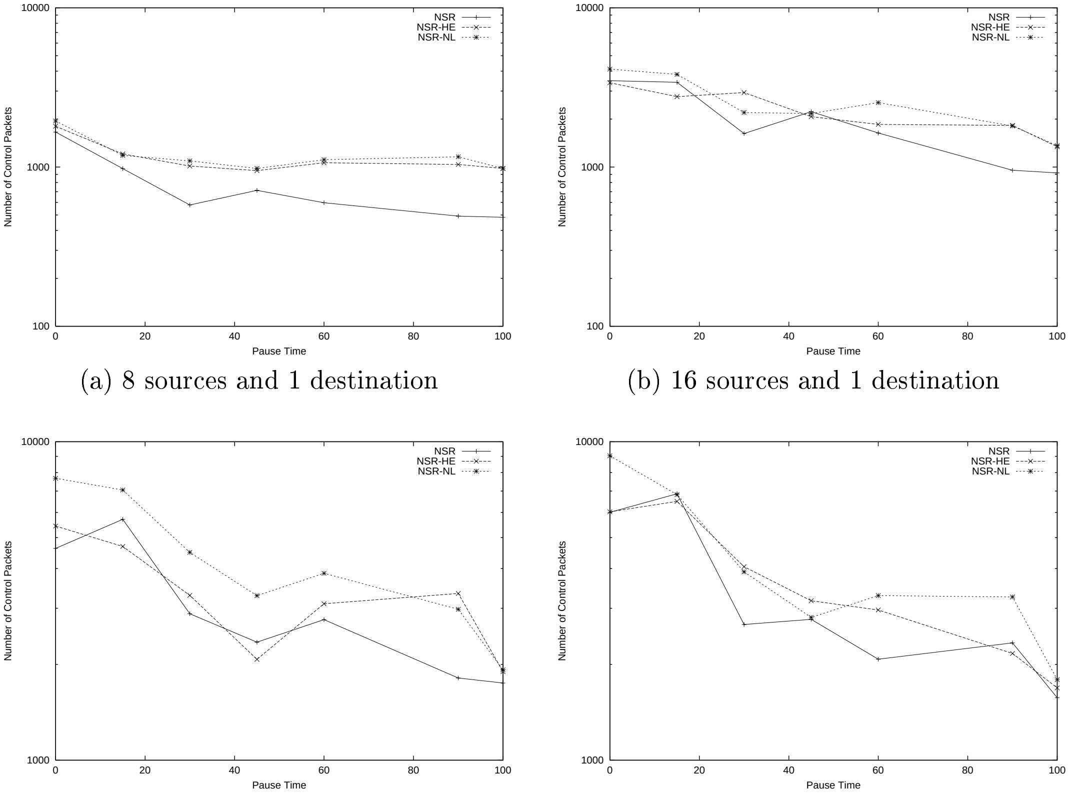
(a) 8 sources and 8 destinations

(b) 16 sources and 16 destinations

(c) 32 sources and 32 destinations

(d) 50 sources and 50 destinations

Figure 4.17: Number of control packets transmitted using the Nsrc-Ndst pattern



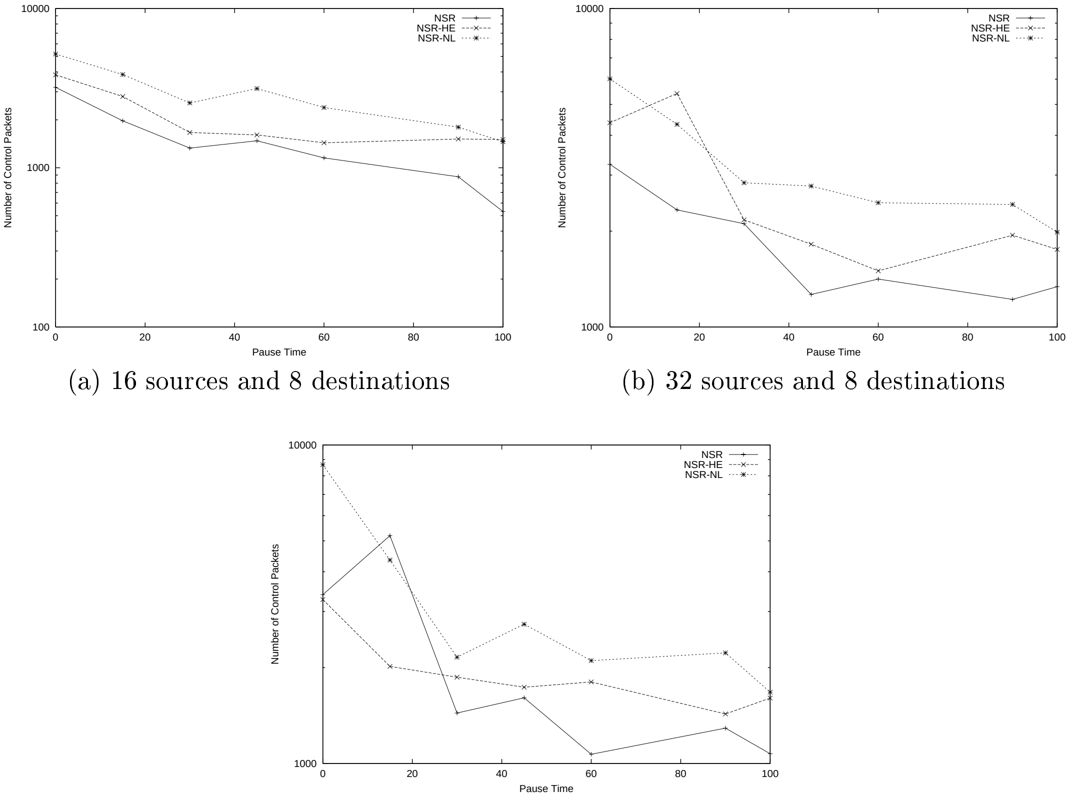
(a) 8 sources and 1 destination

(b) 16 sources and 1 destination

(c) 32 sources and 1 destination

(d) 50 sources and 1 destination

Figure 4.18: Number of control packets transmitted using the Nsrc-1dst pattern

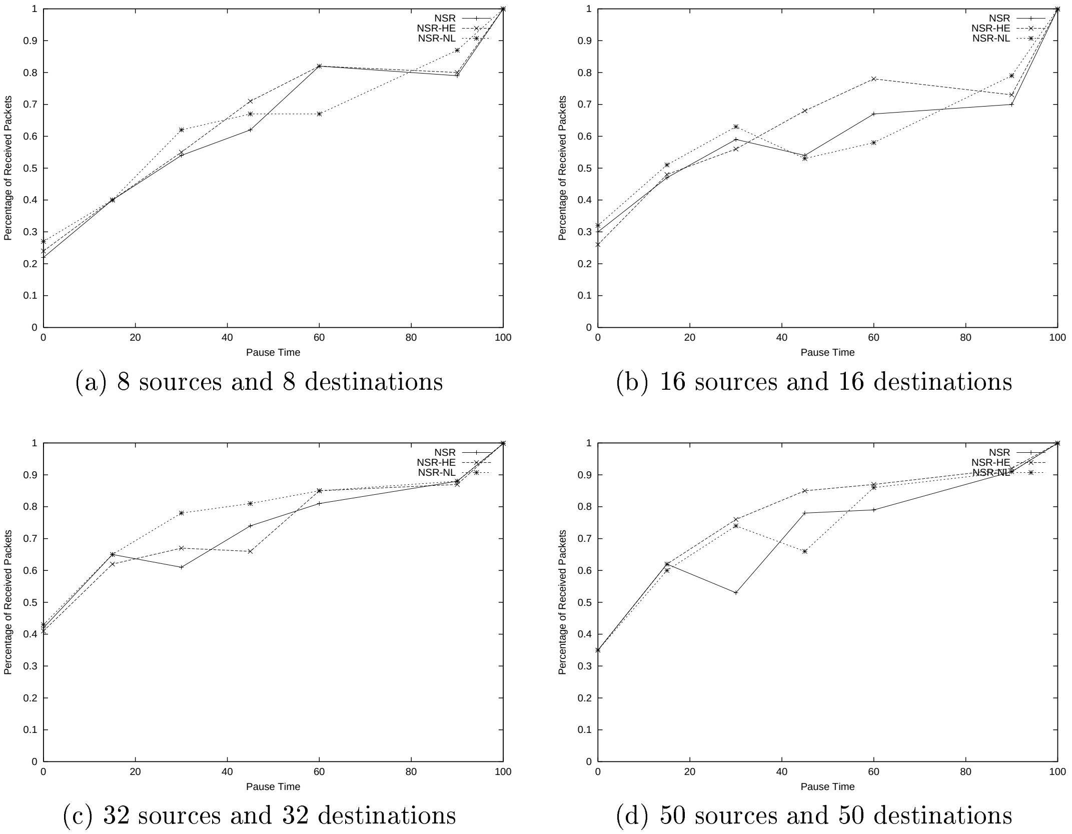


(a) 16 sources and 8 destinations

(b) 32 sources and 8 destinations

(c) 50 sources and 8 destinations

Figure 4.19: Number of control packets transmitted using the Nsrc-8dst pattern



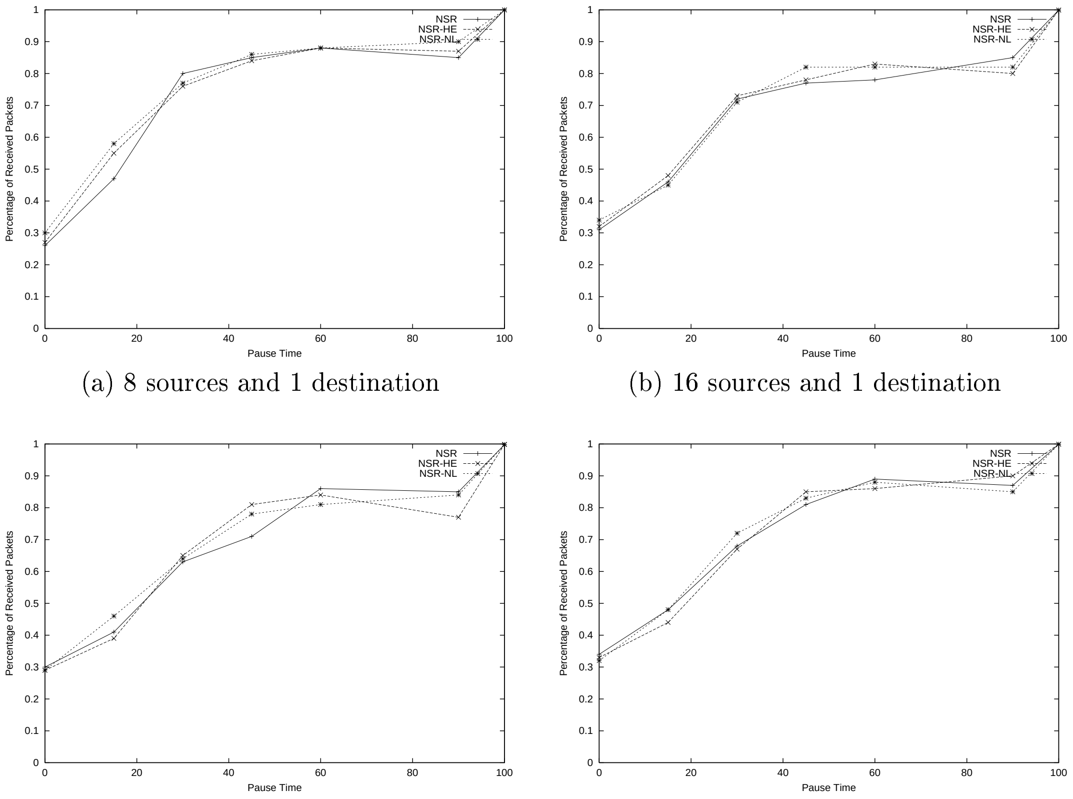
(a) 8 sources and 8 destinations

(b) 16 sources and 16 destinations

(c) 32 sources and 32 destinations

(d) 50 sources and 50 destinations

Figure 4.20: Percentage of data packets received using the Nsrc-Ndst pattern



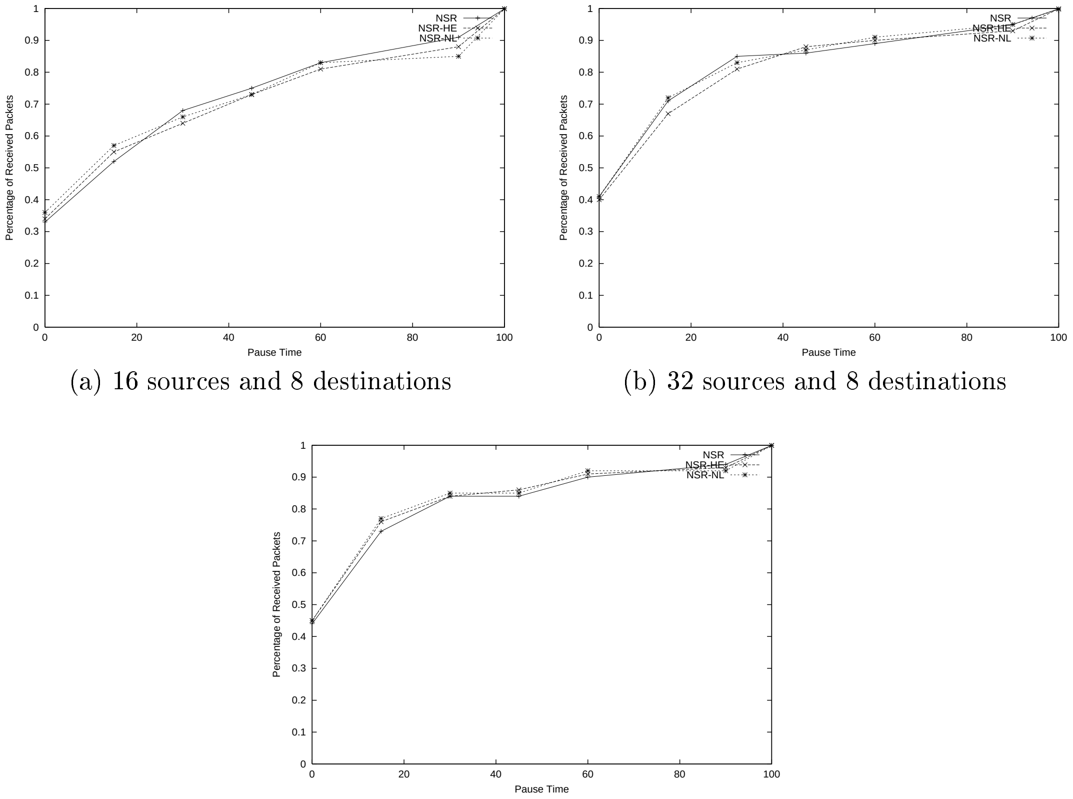
(a) 8 sources and 1 destination

(b) 16 sources and 1 destination

(c) 32 sources and 1 destination

(d) 50 sources and 1 destination

Figure 4.21: Percentage of data packets received using the Nsrc-1dst pattern

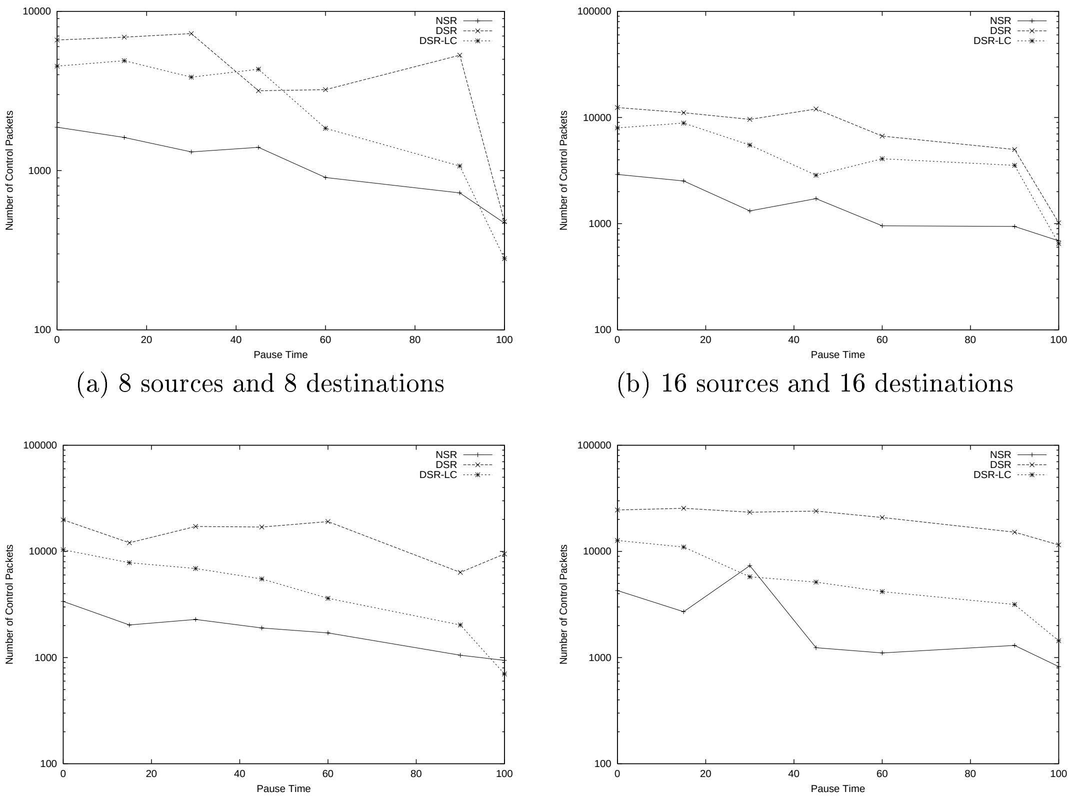


(a) 16 sources and 8 destinations

(b) 32 sources and 8 destinations

(c) 50 sources and 8 destinations

Figure 4.22: Percentage of data packets received using the Nsrc-8dst pattern



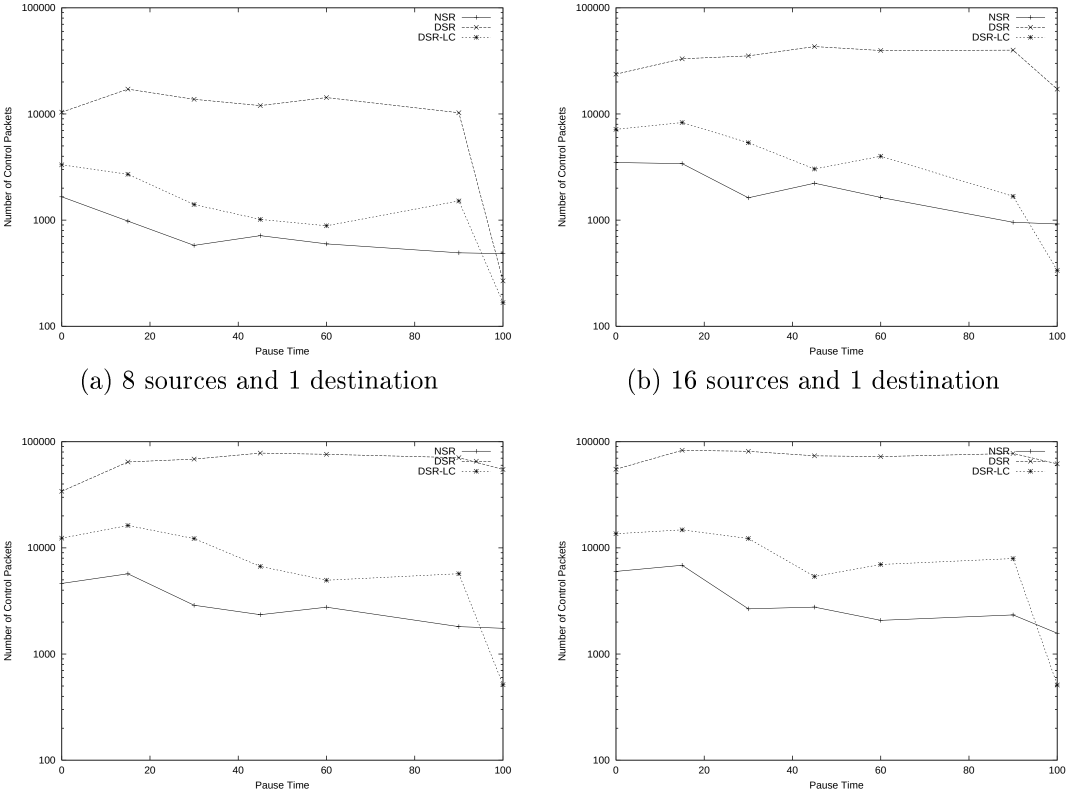
(a) 8 sources and 8 destinations

(b) 16 sources and 16 destinations

(d) 32 sources and 32 destinations

(e) 50 sources and 50 destinations

Figure 4.23: Number of control packets transmitted using the Nsrc-Ndst pattern



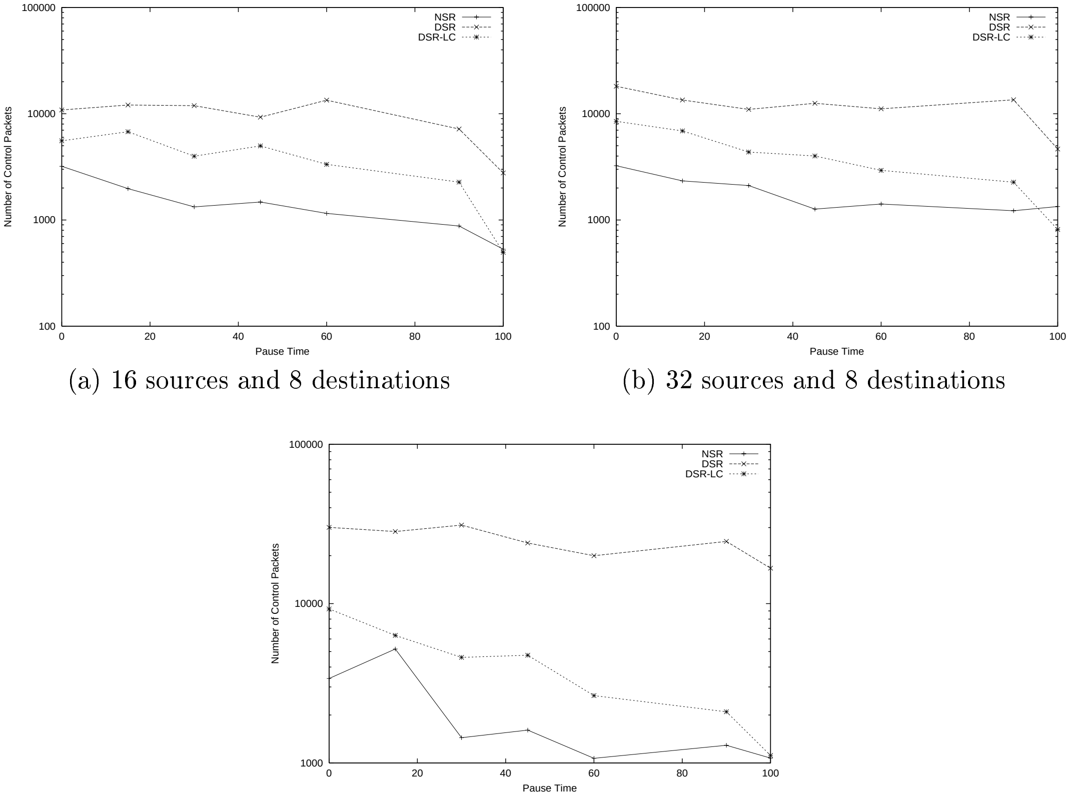
(a) 8 sources and 1 destination

(b) 16 sources and 1 destination

(d) 32 sources and 1 destination

(e) 50 sources and 1 destination

Figure 4.24: Number of control packets transmitted using the Nsrc-1dst pattern

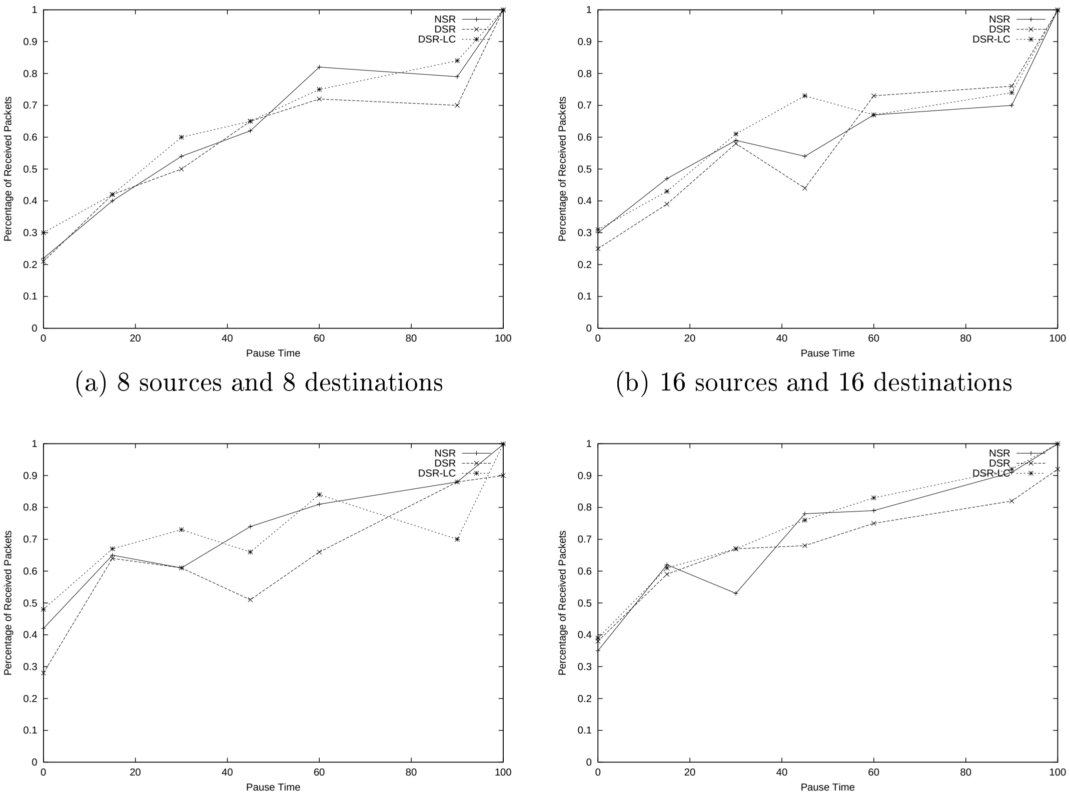


(a) 16 sources and 8 destinations

(b) 32 sources and 8 destinations

(c) 50 sources and 8 destinations

Figure 4.25: Number of control packets transmitted using the Nsrc-8dst pattern



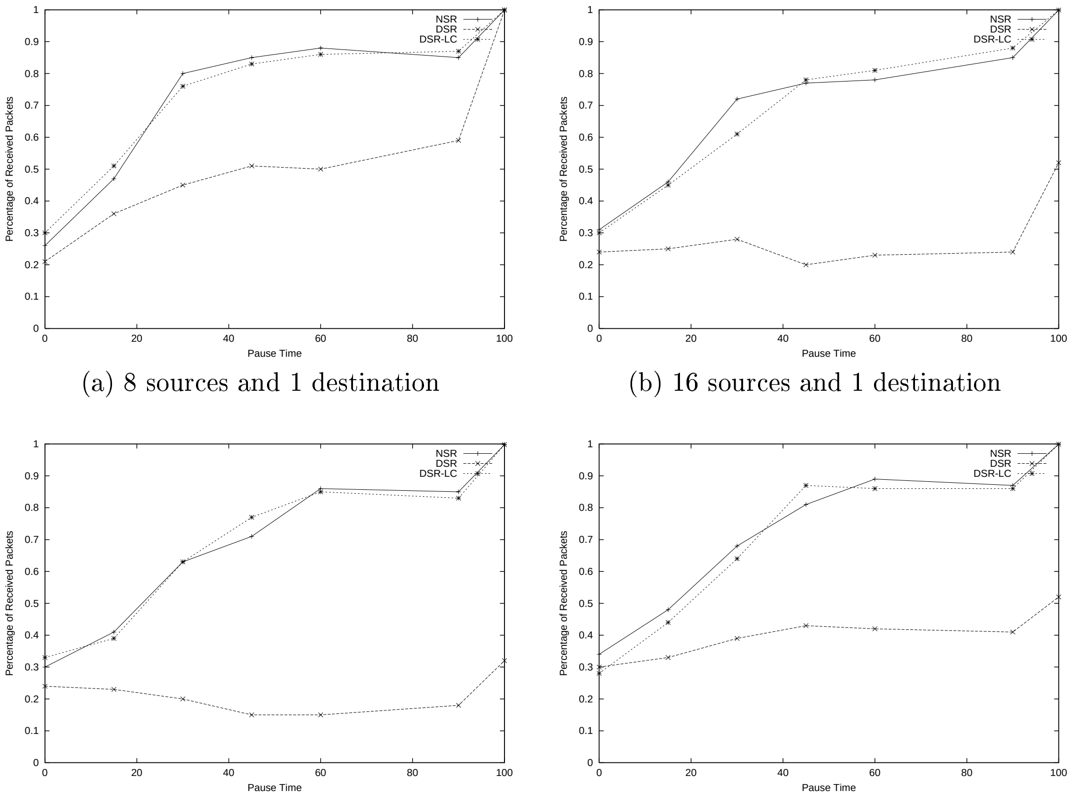
(a) 8 sources and 8 destinations

(b) 16 sources and 16 destinations

(d) 32 sources and 32 destinations

(e) 50 sources and 50 destinations

Figure 4.26: Percentage of data packets received using the Nsrc-Ndst pattern



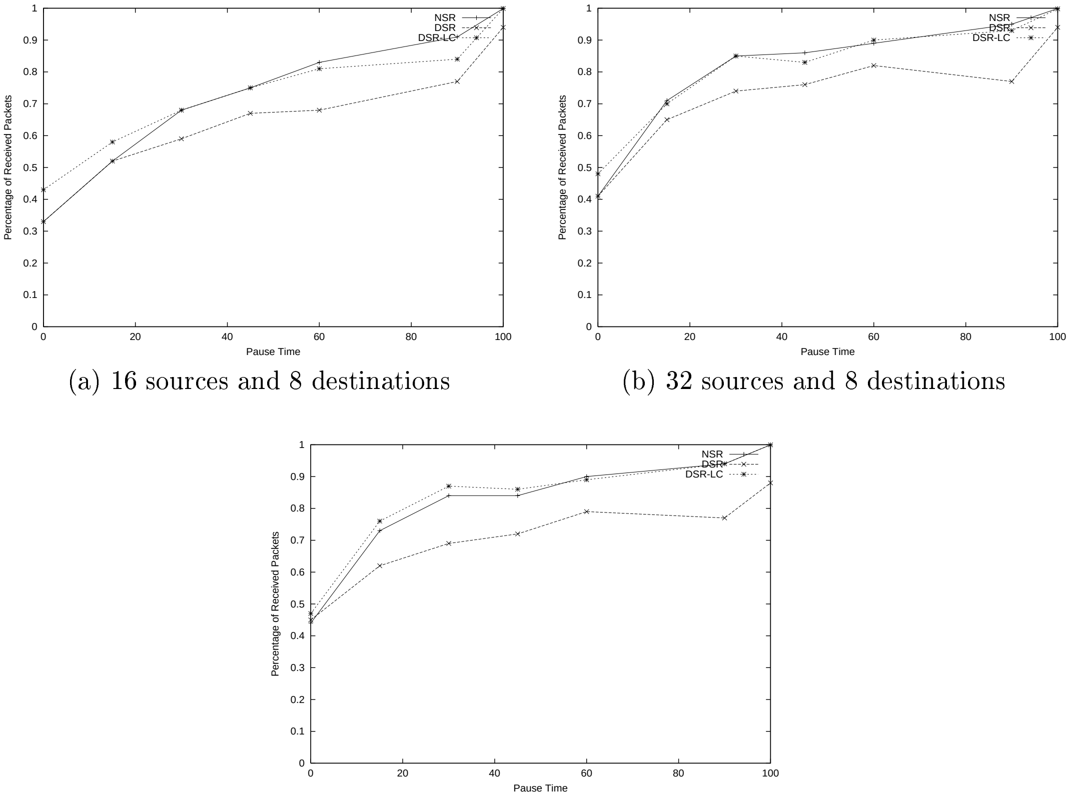
(a) 8 sources and 1 destination

(b) 16 sources and 1 destination

(d) 32 sources and 1 destination

(e) 50 sources and 1 destination

Figure 4.27: Percentage of data packets received using the Nsrc-1dst pattern



(a) 16 sources and 8 destinations

(b) 32 sources and 8 destinations

(e) 50 sources and 8 destinations

Figure 4.28: Percentage of data packets received using the Nsrc-8dst pattern

4.4 Conclusions

We have presented the neighborhood aware source routing protocol (NSR), which we derived from the performance improvements observed in DSR when link caches were used, and the ease with which nodes can inform their neighbors of their own neighbors. The key feature of NSR is that nodes reduce the effort required to fix source routes due to node mobility by using alternate links available in their two-hop neighborhood. Simulations demonstrate the advantages derived from the availability of such alternate paths.

Chapter 5

Summary and Future Work

5.1 Contributions

The goal of this thesis has been to provide robust, efficient, and scalable solutions for unicast routing in both wired and wireless IP networks. We found that link-state information holds the key to achieving this goal. We explored the use of partial link-state information, and made considerable progress in understanding how it can be used to incur low routing overhead while delivering data packets.

Our first routing algorithm, the adaptive link-state protocol (ALP), is targeted to networks based on wired links. Novel features in ALP include using three types of state for any given link to disseminate correctly partial link-state information, and using a designated router per link for each broadcast medium (e.g., a LAN). In ALP, a router sends updates to its neighbors regarding the links in its preferred paths to destinations. Each router decides which links to report to its neighbors based on its local computation of preferred paths. In contrast to LVA, which is the only prior routing algorithm based on selective dissemination of link states, a router does not ask its neighbors to delete links; instead, a router simply updates its neighbors with the most recent information about those links it decides to take out of its preferred paths. Furthermore, when multiple routers are connected through a broadcast medium, they elect distributedly a designated router for each link reported over the broadcast medium; this reduces the number of updates per link sent over a given network. Unlike OSPF, ALP does not require backbones, a designated router does not require the exchange of control packets, and can be used with distributed hierarchical routing schemes proposed in the past for distance-vector routing. Because routers in ALP propagate link-state information selectively, it incurs less communication overhead than algorithms based on topology broadcast. ALP ran successfully in a small testbed implemented with PCs running gateD, and the very same code was used in the reported simulation experiments. Simulations using the actual gateD code for ALP corroborate the fact that ALP achieves the most efficient way of updating routing tables compared to topology broadcast, the distributed Bellman-Ford algorithm, and LVA. ALP addresses the complexity of today's approach to link-state routing by making the computation of routing trees using link-states costs a distributed computation.

We proposed the source-tree adaptive routing (STAR) protocol for multi-hop packet-radio networks that require routers to maintain paths to all destinations in the network. In STAR, a router sends updates to its neighbors regarding the links in its preferred paths to destinations. The links along the preferred paths from a source to each desired destination constitute a *source tree* that implicitly specifies the complete paths from the source to each destination. Each router computes its source tree based on information about adjacent links and the source trees reported by its neighbors, and reports changes to its source tree to all its neighbors incrementally if the link-layer provides reliable broadcast, or atomically otherwise. The aggregation of adjacent links and source trees reported by neighbors constitutes the partial topology known by a router. Prior proposals for link-state routing using partial link-state data without clusters [19, 20] require routers to explicitly inform their neighbors which links they use and which links they stop using. In contrast, because STAR sends only changes to the structure of source trees, and because each destination has a single predecessor in a source tree, a router needs to send only updates for those links that are part of the tree and a single update entry for the root of any subtree of the source tree that becomes unreachable due to failures. Routers receiving a STAR update can infer correctly all the links that the sender has stopped using, without the need for explicit delete updates. STAR incurs smaller communication overhead than the ideal topology broadcast protocol and ALP, and also incurs less overhead than DSR, which is one of the most bandwidth-efficient on-demand routing protocols proposed to date. STAR accomplishes its bandwidth efficiency by: (a) disseminating only that link-state data needed for routers to reach destinations; (b) exploiting that information to ascertain when update messages must be transmitted to detect new destinations, unreachable destinations, and loops; and (c) allowing paths to deviate from the ideal optimum while not creating permanent loops. Because STAR can be used with any clustering mechanism proposed to date, these results clearly indicate that STAR is a very attractive approach for routing in packet-radio networks. The very same code written for the simulation experiments was used in a testbed formed by 40 fixed wireless routers.

Lastly, we proposed the neighborhood aware source routing (NSR) protocol for wireless ad hoc networks with a large number of routers. NSR introduces a new approach to link-state routing in ad hoc networks using on-demand source routing and knowledge of links that exist in the two-hop neighborhood of nodes. In NSR, a node maintains a partial topology of the network consisting of the links to its immediate neighbors (1-hop neighbors), the links to its 2-hop neighbors, and the links in the requested paths to destinations that are more than two hops away. Links are removed from this partial topology graph by aging only, and the lifetime of a link is determined by the node from which the link starts (head node of the link), reflecting with a good degree of certainty the variations in mobility of the node. The key feature of NSR is that nodes reduce the effort required to fix source routes due to node mobility by using alternate links available in their two-hop neighborhood. Simulations demonstrate the advantages derived from the availability of such alternate paths. The simulation results indicate that NSR incurs far less communication overhead while delivering packets with the same or better delivery rates and average delays as DSR using either path caches or link caches. Perhaps, more importantly, NSR scales well for a large number of routers and varied data workload.

The text of this dissertation includes material that has previously been published in [20], [21], [22], [23], [24], and [53]. The co-author listed in these publications directed and supervised the research which forms the basis for this dissertation.

5.2 Future Work

One important area for future research is the optimization of local path computations. A major drawback of link-state protocols is that they need to run computationally expensive shortest-path algorithm at every router. Although the smaller topology graph size in ALP and STAR improves the running time for the shortest-path algorithm, the local computations are still far more complex than those needed in distance-vector algorithms. An ideal solution for the local computations would provide for incremental changes in the source graph, rather than a complete rebuilding after any change.

Multimedia applications using the Internet create a demand for quality of service (QoS) guarantees, such as a guaranteed bandwidth and delays. To be able to support QoS, congestion control and avoidance need to be integrated with the routing decisions. Extending ALP and STAR to provide multiple loop-free paths to the same destination, as well as computing such paths based on multiple constraints (e.g., delay, bandwidth, jitter) are research topics that need to be addressed.

Large populations of mobile nodes can be supported in an ad hoc network by accepting longer route acquisition latencies in an on-demand protocol. Starting from a network without structure or valid routes, the minimum route acquisition latency that allows full connectivity is the product of the maximum diameter of the network multiplied by the minimum node traversal time for route requests. The average route acquisition latency at any given time can be much worse than this depending on the dissemination of valid route information and network congestion [5]. An obvious extension of the work reported on NSR would be to investigate how it scales with the number of routers, link-layer capacity, node mobility, area of coverage, and data traffic pattern.

As the global Internet evolves we may find topologies formed by multi-hop wireless networks with fixed routers (anchors) offering connectivity to the Internet for mobile routers and hosts. For this type of networks, a hybrid approach of table-driven with on-demand routing might result in the best performing solution. For instance, given that most of the data traffic would be with destinations not belonging to the wireless network, the routes from each mobile node to the anchor nodes, and vice versa, might be updated in a proactive way, and the routes between mobile nodes might be updated in a reactive form. An interesting venue of research is to study how the routing mechanisms of STAR and NSR could be combined to provide an efficient routing solution for such topologies. Several other areas invite further investigation in multi-hop wireless networks, including: routing solutions to provide quality of service guarantees to applications, secure routing, and multicast routing.

Bibliography

[1] R. Albrightson, J.J. Garcia-Luna-Aceves, and J. Boyle. EIGRP-A Fast Routing Protocol Based on Distance Vectors. *Proc. Networld/Interop 94*, May 1994.

[2] J. Behrens and J. J. Garcia-Luna-Aceves. Hierarchical Routing Using Link Vectors. *Proc. IEEE INFOCOM*, March 1998.

[3] D. Bertsekas and R. Gallager. *Data Networks. 2nd Ed.* Prentice-Hall, 1992.

[4] D. Beyer. *The C++ Protocol Toolkit - Reference Manual.* Rooftop Communications, 1998.

[5] Ed. C. Perkins. *Ad Hoc Routing.* Addison-Wesley Longman, 2001.

[6] C. Cheng and et. al. A Loop-Free Extended Bellman-Ford Routing Protocol Without Bouncing Effect. *Proc. of ACM SIGCOMM*, 1989.

[7] IEEE Computer Society LAN MAN Standards Committee. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE Std 802.11*, 1997.

[8] Merit GateD Consortium. GateD Documentation. *http://www.gated.org*, 1998.

[9] S. Deering and R. Hinden. Internet Protocol Version 6 (IPv6) Specification. *RFC*, 2460, 1998.

[10] M. Steenstrup (Ed.). *Routing in Communication Networks.* Prentice-Hall, 1995.

[11] J. Broch et al. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. *Proc. ACM MOBICOM*, October 1998.

[12] J.J. Garcia-Luna-Aceves et al. Wireless Internet Gateways (WINGS). *MILCOMM*, November 1997.

[13] R. Dube et al. Signal Stability-Based Adaptive Routing (SSA) for Ad-Hoc Mobile Networks. *IEEE Personal Communications Magazine*, February 1997.

[14] S. Basagni et al. A Distance Routing Effect Algorithm for Mobility (DREAM). *Proc. ACM MOBICOM*, pages 76–84, October 1998.

[15] K. Fall and K. Varadhan. ns Notes and Documentation. *The VINT Project, UC Berkeley, LBL, USC/ISI and Xerox PARC*, http://www.isi.edu/nsnam/ns, 1999.

[16] C. L. Fullmer and J.J. Garcia-Luna-Aceves. Solutions to Hidden Terminal Problems in Wireless Networks. *Proc. of ACM SIGCOMM*, September 1997.

[17] E. Gafni. Generalized Scheme for Topology-Update in Dynamic Networks. *G. Goos and J. Hartmanis, Eds.*, 312:187–196, 1987.

[18] J.J. Garcia-Luna-Aceves. Loop-Free Routing Using Diffusing Computations. *IEEE/ACM Trans. Networking*, 1:130–141, February 1993.

[19] J.J. Garcia-Luna-Aceves and J. Behrens. Distributed, scalable routing based on vectors of link states. *IEEE Journal on Selected Areas in Communications*, 13, October 1995.

[20] J.J. Garcia-Luna-Aceves and M. Spohn. Scalable Link-State Internet Routing. *Proc. International Conference on Network Protocols*, October 1998.

[21] J.J. Garcia-Luna-Aceves and M. Spohn. Efficient Routing in Packet-Radio Networks Using Link-State Information. *Proc. IEEE WCNC*, September 1999.

[22] J.J. Garcia-Luna-Aceves and M. Spohn. Source-Tree Routing in Wireless Networks. *Proc. International Conference on Network Protocols*, October 1999.

[23] J.J. Garcia-Luna-Aceves and M. Spohn. Transmission-Efficient Routing in Wireless Networks using Link-State Information. *ACM Mobile Networks and Applications Journal*, accepted for publication in Special Issue on Energy Conserving Protocols in Wireless Networks, 1999.

[24] J.J. Garcia-Luna-Aceves and M. Spohn. *Bandwidth-Efficient Link-State Routing in Wireless Networks.* Ad Hoc Routing (C. Perkins, Ed.), Chapter 10, Addison-Wesley Longman, 2001.

[25] Z. Haas and M. Pearlman. The Zone Routing Protocol for Highly Reconfigurable Ad-Hoc Networks. *Proc. of ACM SIGCOMM*, August 1998.

[26] C. Hedrick. Routing Information Protocol. *RFC*, 1058, June 1988.

[27] Yih-Chun Hu and D. Johnson. Caching Strategies in On-Demand Routing Protocols for Wireless Ad Hoc Networks. *Proc. ACM MOBICOM*, 2000.

[28] J.M. Jaffe. Algorithms for Finding Paths with Multiple Constraints. *Networks*, 14:95–116, 1984.

[29] J.J. Garcia-Luna-Aceves and A. Tzamaloukas. Reversing the Collision Avoidance Handshake in Wireless Networks. *Proc. ACM MOBICOM*, August 1999.

[30] D. Johnson and D. Maltz. Protocols for Adaptive Wireless and Mobile Networking. *IEEE Personal Communications Magazine*, 3(1), February 1996.

[31] J. Jubin and J. Tornow. The DARPA Packet Radio Network Protocols. *Proceedings of the IEEE*, 75(1), January 1987.

[32] L. Kleinrock and F. Kamoun. Hierarchical Routing for Large Networks: Performance Evaluation and Optimization. *Computer Networks*, 1:155–174, 1977.

[33] Y-B. Ko and N. Vaidya. Location-Aided Routing (LAR) in Mobile Ad Hoc Networks. *Proc. ACM MOBICOM*, pages 66–75, October 1998.

[34] V.O.K. Li and R. Chang. Proposed Routing Algorithms for the US Army Mobile Subscriber Equipment (MSE) Network. *MILCOMM*, October 1986.

[35] J. Moy. OSPF Version 2. *RFC*, 2328, 1998.

[36] S. Murthy and J.J. Garcia-Luna-Aceves. An Efficient Routing Protocol for Wireless Networks. *ACM Mobile Networks and Application Journal, Special issue on Routing in Mobile communications Networks*, 1996.

[37] S. Murthy and J.J. Garcia-Luna-Aceves. Loop-Free Internet Routing Using Hierarchical Routing Trees. *Proc. IEEE INFOCOM*, April 1997.

[38] S. Murthy and J.J. Garcia-Luna-Aceves. A Loop-Free Routing Protocol for Large-Scale Internets Using Distance Vectors. *Computer Communications (Elsevier)*, 21(2):147–161, 1998.

[39] International Standards Organization. Intra-Domain IS-IS Routing Protocol. *ISO/IEC JTC1/SC6 WG2 N323*, September 1989.

[40] V. Park and M. Corson. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. *Proc. IEEE INFOCOM*, April 1997.

[41] C. Perkins and P. Bhagwat. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. *Proc. of ACM SIGCOMM*, October 1994.

[42] C. Perkins and E. Royer. Ad-Hoc On-Demand Distance Vector Routing. *Proc. of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, February 1999.

[43] R. Perlman. Fault-Tolerant Broadcast of Routing Information. *Computer Networks and ISDN*, 7:395–405, 1983.

[44] J. Postel. Internet Protocol. *RFC*, 791, 1981.

[45] The CMU Monarch Project. Wireless and Mobility Extensions to ns-2 - Snapshot 1.0.0-beta. *http://www.monarch.cs.cmu.edu/cmu-ns.html*, August 1998.

[46] M. Pursley and H.B. Russell. Routing in Frequency-Hop Packet Radio Networks with Partial-Band Jamming. *IEEE Trans. Commun.*, 41(7):1117–1124, 1993.

[47] B. Rajagopalan and M. Faiman. A New Responsive Distributed Shortest-Path Routing Algorithm. *Proc. of ACM SIGCOMM*, September 1989.

[48] C.V. Ramamoorthy and W. Tsai. An Adaptive Hierarchical Routing Algorithm. *Proc. IEEE COMPSAC*, pages 93–104, November 1983.

[49] R. Ramanathan and M. Steenstrup. Hierarchically-Organized, Multihop Mobile Wireless Networks for Quality-of-Service Support. *Proc. ACM Mobile Networks and Applications*, 3(1):101–119, 1998.

[50] Y. Rekhter. Inter-Domain Routing Protocol (IDRP). *Internetworking: Research and Experience*, 4(2):61–80, June 1993.

[51] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). *RFC*, 1644, 1994.

[52] S. Murthy. *Routing in Packet-Switched Networks Using Path-Finding Algorithms, Ph.D. Thesis.* University of California, Santa Cruz, September 1996.

[53] M. Spohn and J.J. Garcia-Luna-Aceves. Neighborhood Aware Source Routing. *To appear in Proc. of ACM Symposium on Mobile Ad Hoc Networking and Computing*, October 2001.

[54] Z. Tang and J.J. Garcia-Luna-Aceves. A Protocol for Topology-Dependent Transmission Scheduling. *Proc. IEEE WCNC*, September 1999.

[55] C-K. Toh. *Wireless ATM & Ad-Hoc Networks.* Kluwer, 1996.

[56] C. Zhu and S. Corson. A Five Phase Reservation Protocol (FPRP) for Mobile Ad-Hoc Networks. *Proc. IEEE INFOCOM*, 1998.